NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

IMPLEMENTATION OF A NETWORK ADDRESS TRANSLATION MECHANISM OVER IPV6

by

Trevor J. Baumgartner
Matthew D.W. Phillips

June 2004

Thesis Advisor:	Cynthia E. Irvine
Co-Advisor:	Thuy D. Nguyen

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.		
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2004	**3. REPORT TYPE AND DATES COVERED** Master’s Thesis
4. TITLE AND SUBTITLE: Implementation of a Network Address Translation Mechanism Over IPv6		**5. FUNDING NUMBERS**
6. AUTHOR(S)Trevor J. Baumgartner & Matthew D.W. Phillips		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.		
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Network Address Translation (NAT) for IPv4 was developed primarily to curb overcrowding of the Internet due to dwindling global IP addresses; however, NAT provides several other benefits. NAT can be used to mask the internal IP addresses of an Intranet. IPv6, the emerging standard for Internet addressing, provides three times the number of bits for IP addressing. While IPv6 does not need NAT for connectivity, other NAT features such as address hiding are valuable. There is currently no NAT implementation for IPv6.

The focus of this research was the design and development of a NAT implementation for IPv6. This implementation will be used within a multilevel testbed. In addition, the NAT implementation developed here can facilitate the Department of Defense (DoD) transition to IPv6 planned for 2008 by providing services currently not available for IPv6.

A working implementation of NAT for IPv6 within the Linux kernel has been produced. The NAT development created here has been tested for support of the protocols of TCP, UDP and ICMP for IPv6.

14. SUBJECT TERMS Network Address Translation, NAT, IPv6, IPv4, MYSEA, MLS, Common Criteria, Linux Source Code, Netfilter, Iptables, Ip6tables			**15. NUMBER OF PAGES** 270
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

THIS PAGE INTENTIONALLY LEFT BLANK

IMPLEMENTATION OF NETWORK ADDRESS TRANSLATION MECHANISM OVER IPV6

Trevor J. Baumgartner
Ensign, United States Navy
B.S., United States Naval Academy, 2003

Matthew D.W. Phillips
Ensign, United States Navy
B.S., United States Naval Academy, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2004

Authors: Trevor J. Baumgartner

Matthew D.W. Phillips

Approved by: Dr. Cynthia E. Irvine
Thesis Advisor

Thuy D. Nguyen
Co-Advisor

Dr. Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Network Address Translation (NAT) for IPv4 was developed primarily to curb overcrowding of the Internet due to dwindling global IP addresses; however, NAT provides several other benefits. NAT can be used to mask the internal IP addresses of an Intranet. IPv6, the emerging standard for Internet addressing, provides three times the number of bits for IP addressing. While IPv6 does not need NAT for connectivity, other NAT features such as address hiding are valuable. There is currently no NAT implementation for IPv6.

The focus of this research was the design and development of a NAT implementation for IPv6. This implementation will be used within a multilevel testbed. In addition, the NAT implementation developed here can facilitate the Department of Defense (DoD) transition to IPv6 planned for 2008 by providing services currently not available for IPv6.

A working implementation of NAT for IPv6 within the Linux kernel has been produced. The NAT development created here has been tested for support of the protocols of TCP, UDP and ICMP for IPv6.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AH	Authentication Header
ALG	Application Level Gateway
CC	Common Criteria
CM	Configuration Management
COTS	Commercial Off-the-shelf
DARPA	Defense Advanced Research Projects Agency
DNAT	Destination Network Address Translation
DoD	Department of Defense
EAL	Evaluation Assurance Level
ESP	Encapsulating Security Protocol
FTP	File Transfer Protocol
ICMP	Internet Control Message Protocol
IKE	Internet Key Exchange
IP	Internet Protocol
IPsec	Internet Protocol security
IPSO	Internet Protocol Security Options
IPv4	Internet Protocol version four
IPv6	Internet Protocol version six
IT	Information Technology
LAN	Local Area Network
MLS	Multilevel Security
MTU	Maximum Transmission Unit
MYSEA	Monterey Security Architecture
NAPT	Network Address Port Translation
NAT	Network Address Translation
NCW	Network Centric Warfare
PP	Protection Profile
RFC	Request For Comments
SNAT	Source Network Address Translation
ST	Security Target
TCM	Trusted Channel Modules
TCP	Transmission Control Protocol
TOE	Target of Evaluation
TPE	Trusted Path Extension
TSF	Target of Evaluation Security Functions
UDP	User Datagram Protocol
VPN	Virtual Private Network

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

We would like to thank our thesis advisor, Dr. Cynthia Irvine, for the motivation and immense assistance in completing this thesis. We appreciate the technical guidance and support given through all stages of this endeavor.

Thanks also to our second advisor Thuy Nguyen for putting in long, hard hours to assist us. She went above and beyond expectations by spending many hours at home and on the weekends in the lab, in order to ensure this project was completed.

Many thanks to Jean Khosalim for his Linux expertise and his assistance. He also spent many hours on the weekends in the lab to assist us in our work. Without him we would still be stuck in a kernel trap.

Amy and Jennifer, we would like to thank you for all of the love and support you have given us. Without your patience and understanding we would never have been able to complete this thesis.

Finally, we would like to thank our parents for everything they have given us. The love, support, and understanding have made us who we are today. Thank you again for everything.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Internet Protocol version four (IPv4), was accepted for military network use by the Department of Defense in 1981. [IP] At the time of its inception, the DARPA Net was a connectivity testbed. As it expanded and its popularity grew, it became the commercialized Internet still in use today. This expansive network is founded on a host-based addressing architecture that assigns a 32-bit address to each connected system. In the 1980’s, the expandability of the 32-bit address used in IPv4 was not a consideration due to the limited use of the protocol, primarily for government and academic purposes. Now, with the expanding growth of the Internet, it is said that the IPv4 address space will be outdated by 2010. [NGI] This imminent address exhaustion drives the need for a new protocol that will allow for a greater number of addresses and a modular approach to security.

Network Address Translation (NAT) was introduced as a temporary solution to the rapidly overcrowding address space in IPv4. NAT allows an entire network of systems to use a single IP address or pool of IP addresses to access the external Internet. The NAT mechanism does this by replacing the true source address of the internal system with the border address in all outgoing datagrams. Furthermore, the mechanism tracks the connection between internal and external systems in order to maintain addressing information for all incoming datagrams.

Internet Protocol version six (IPv6) is the solution to the address space problem. Instead of 32 bits for addressing, this new protocol uses 128 bits, allowing for a theoretical maximum of 2^{128} addresses. Since Network Address Translation (NAT) was designed to reduce overcrowding in IPv4, many believe that this functionality will not be needed with IPv6. The purpose of this thesis is to provide evidence that certain benefits provided only by NAT are still necessary as well as to create a working implementation of it.

A. PURPOSE OF STUDY

The popular belief is that since overcrowding is not an issue in IPv6, NAT functionality will not be needed. However, NAT has two key functionalities aside from address space expansion that are beneficial from a security perspective. First, by using the NAT mechanism, one is able to mask the IP addresses of internal systems. NAT does this by replacing the source address of an outgoing datagram with another address from a pool of IP addresses or a single constant IP address. The NAT device keeps track of this connection and alters all incoming datagrams destined to the border NAT address to reflect the true internal IP address. Second, the NAT mechanism also hides the internal structure of an intranet since all connections to the Internet must first pass through the NAT border device. This forces all external devices to only detect the NAT border device: it is not possible to diagram the internal topology of the network.

It is for these security benefits that this research is being conducted. The goals of this research are two-fold. First, the benefits, drawbacks and feasibility of an IPv6 NAT implementation were examined. This is advantageous to both this thesis in providing direction as well as to future research by providing a solid framework of background information, design implementation and future recommendations. Second, this project has produced a working implementation for NAT over IPv6. This implementation was done through a modified Linux 2.6.5 kernel designed to support connection tracking in IPv6.

There exist multiple benefits of this research. It contributes to the DoD initiative to transition to IPv6 from IPv4 by FY2008. [MEMO] Also, research conducted into IPv6 transition mechanisms will aid the construction of hybrid networks that support both IPv4 and IPv6 to ease the eventual transition to IPv6. This research also supports the Network Centric Warfare (NCW) model and shipboard operations by providing the network security benefit of address hiding and internal network structure masking. NAT can also be used to reduce the cost of leasing a range of IP addresses by allowing an entire LAN to operate on as few as one leased IP address. Finally, NAT for IPv6 contributes to the implementation of high assurance multilevel security systems, such as MYSEA, for use by coalitions through its application in a multilevel testbed.

B. OVERVIEW OF CHAPTERS

This section contains a brief overview of the subsequent chapters.

1. Chapter II, "Network Address Translation Protocol in IPv4"

This chapter provides background information on both IPv4 and NAT. The first part discusses the IPv4 protocol including general background information, its header structure, security issues and the addressing scheme. The second part of the chapter is to familiarize the reader with NAT by explaining the mechanisms used by the multiple types of NAT and the benefits of each.

2. Chapter III, “Internet Protocol, Version 6”

Chapter III explains the background of IPv6 as well as its header format, addressing scheme and address allocation. This chapter also explores the existing security features within IPv6 in addition to emerging technologies. Furthermore, it contains a comparison between the networking and security features provided by NAT for IPv6 and the security features provided by NAT that are desired but not provided by IPv6. Finally, this chapter describes the existing IPv6 support within the current Linux 2.6.5 kernel.

3. Chapter IV, “Monterey Security Architecture”

This chapter explains the necessity for systems to provide multilevel security and the creation of the Monterey Security Architecture (MYSEA) to address those needs. It outlines the design of all relevant components within the architecture and their implementation within the IPv6 NAT testbed.

4. Chapter V, “Common Criteria Assurance Level Exploration”

Chapter V provides the reader with background regarding the Common Criteria. It also describes the evaluation process for IT products and explores the requirements necessary for an assurance evaluation at EAL5. This is done through the framework of the IPv6 NAT implementation created in this project.

5. Chapter VI, “Development of NAT in IPv6”

This chapter summarizes the development process used to implement NAT for IPv6 in conjunction with this thesis. It explains the methodology used to port the existing IPv4 NAT code for use with IPv6. It details the major programming difficulties encountered during the porting process and how they were resolved. It also explains the debugging process used, as well as functionality testing of the resulting implementation.

6. Chapter VII, “Conclusion”

Chapter VII gives an analysis of the IPv6 NAT implementation as it is integrated within the Linux kernel. It also provides design implementation ideas for possible future developments.

THIS PAGE INTENTIONALLY LEFT BLANK

II. NETWORK ADDRESS TRANSLATION IN IPV4

Network Address Translation (NAT) has served to increase available IP address space as originally noted in 1994 [TNAT]. This chapter contains a summary of current NAT implementations and the functionalities provided by NAT. The chapter then examines NAT and related functionalities, as implemented in Red Hat 9.0, the Linux platform on which NAT for IPv6 will be developed. Since NAT has not yet been developed for IPv6, any reference to NAT, unless explicitly stated, refers to NAT for IPv4.

A. BACKGROUND AND ANALYSIS

This section presents an overview of IPv4, its structure and addressing scheme. This section also provides an overview of NAT.

1. Internet Protocol, Version 4

a. Background Information

Today's current Internet Protocol, Version 4 (IPv4) was specified in 1981 with RFC 791. [IP] The IP protocol resides at layer 3 of the OSI 7-layer model (see Figure 1) which is responsible for the management of network connections. [OSI]

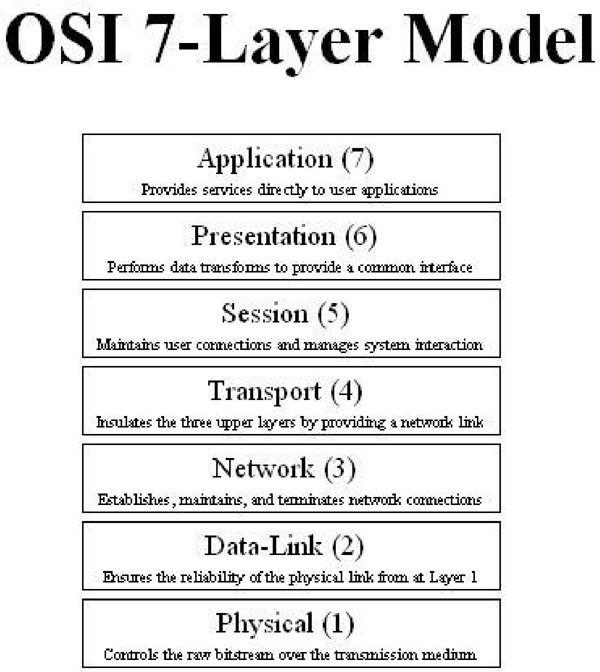

OSI 7-Layer Model
Application (7)
Provides services directly to user applications
Presentation (6)
Performs data transforms to provide a common interface
Session (5)
Maintains user connections and manages system interaction
Transport (4)
Insulates the three upper layers by providing a network link
Network (3)
Establishes, maintains, and terminates network connections
Data-Link (2)
Ensures the reliability of the physical link from at Layer 1
Physical (1)
Controls the raw bitstream over the transmission medium

Figure 1. The OSI 7-layer Model [OSI]

The purpose of layer 3, or where IP services are implemented, was to allow hosts on different network topologies to have a standard means of transporting data packets to each other across the Internet. Each host would have a unique IP address, almost like a mailing address, to distinguish it from all of the other hosts connected to the Internet. IPv4 performs two main functions: addressing and fragmentation. The purpose for addressing is obvious enough, because without a unique address routers would be unable to determine the intended destination for each packet. Fragmentation may not seem as necessary until one realizes the myriad of networks and respective standards that exist. Ethernet has a maximum transmission unit (MTU) size of 1492 bytes, while a Token Ring can be configured to have an MTU of 2046. Other layer 2 protocols have other MTU sizes. [MTU] Thus, for a Token Ring packet to traverse an Ethernet topology, it must be fragmented into two Ethernet packets. In addition to addressing and fragmentation, IP allows for error reporting through the use of the Internet Control Message Protocol (ICMP). [ICMP SPEC]

b. IP Header Structure

IP headers contain all of the IP addressing, transportation, and processing information for each packet. The IP header is preceded by a layer two header and information dependant upon the networking standard (Ethernet, Token Ring, etc.), and is followed by the packet payload. Figure 2 below, from RFC 791, displays an example IPv4 header.

```
 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2. IPv4 Header

The 192 bit header above displays both the bit count and the respective field name. If laid out in a sequential, linear fashion the header would read from left to right, top to bottom. What follows is the bit length of each field and a description, taken verbatim from RFC 791 [IP]:

- Version (4 bits): The Internet Protocol version.

- IHL (4 bits): Internet Header Length; length of the header in 32 bit words, indicating where the data begins.

- Type of Service (8 bits): Indicates the abstract quality of service parameters desired for this packet.

- Total Length (16 bits): Length of the entire packet (including header) in bytes.

- Identification (16 bits): A number assigned by the sender to help with re-assembling fragmented packets.

- Flags (3 bits): Control flags; Reserved (must be 0), Don’t Fragment (0 means May Fragment), More Fragments (0 means Last Fragment).

- Fragment Offset (13 bits): Indicates where, in the un-fragmented packet, this fragment belongs; measured in units of 64 bits, the first fragment has offset zero.

- Time To Live (8 bits): TTL; indicates the number of times a packet may be processed before being destroyed; it is decremented by one every time it is processed by a host, router, etc.; when it reaches zero the packet is destroyed.

- Protocol (8 bits): Specifies the OSI layer four (next level) protocol in the payload following the IP header (ie, TCP, FTP, etc.).

- Header Checksum (16 bits): A checksum on the IP header only; it is recomputed every time any of the header values are altered.

- Source Address (32 bits): IP address where the packet came from.

- Destination Address (32 bits): IP address where the packet is ultimately destined.

- Options (varies): Various options for the IP packet; its length varies because many of the options have a varying size; the options field may require padding so that it ends on a 32 bit boundary. Below is a list of available options:

  + Security – Security and compartmentation information

  + Loose Source Routing – Specifies a route that, at some point, must be followed (other nodes may be stopped at as well).

  + Strict Source Routing – Specifies a route that must be exactly followed with no other nodes stopped at.

  + Record Route – Record the IP address of each node that processes the packet.

  + Stream ID – Carries a 16-bit SATNET stream identifier through networks not supporting the stream concept.

  + Internet Timestamp – Each forwarding node inserts a timestamp into this field.

c. Security

The IPv4 standard relies on applications and upper-level protocols to implement security features. The security option described in the last section only provides compartmentalization as a method of security and this is only effective if systems that process the packet adhere to the standard. The protocol allows the options field to contain information regarding the intended compartment of the packet. This field is for administrative purposes only and does not support encryption or data security services. However, these labels can support, as noted by the DoD Internet Protocol Security Options (IPSO), a classification scheme that enables packets to be labeled in a Multi-level Secure (MLS) environment. As stated in RFC 1108, this labeling system is designed for a classification system rather than a cryptographic system. [DoD SOIP]

d. Addressing

IP addresses are 32 bits long and can be represented in either bitwise or dotted-decimal notation. Figure 3 gives an example of this:

```
10000000.00001111.11111111.00000000 = 128.15.255.0
```

Figure 3. Bitwise / Dotted-decimal

By using 32 bits for its address space, IPv4 is limited to slightly more than 4.2 billion unique IP addresses, which at the time of its conception was thought to be sufficient; however, the world-wide Internet boom quickly depleted IP addresses to the point that solutions to the dwindling number of addresses had to be found. IPv4 has a addressing scheme that declared networks to be of three different sizes, or *classes*. Class A networks, the largest but also the least abundant, use the first 8 bits of the 32 addressing bits for network identification and the last 24 bits for host identification. Class B networks use the first 16 bits for network identification, while Class C networks use the first 24 bits. [IP]

Unfortunately, this scheme wastes addresses. Assume, for example, that a software company is given a class C address for its 100 computers. This would leave 156 IP addresses unused by the company. To prevent such waste, IPv4 also uses a classless addressing scheme, which essentially creates networks using any number of leading bits through a subnet mask. The subnet mask allows the class to be partitioned by reserving a portion of the host address to reference the underlying subnets created by the division of the address space. Another important aspect of the IPv4 addressing scheme, defined by RFC 1918 [AAPI], is the reservation of certain ranges of addresses for private networking. These private network addresses are not routable and cannot be used on the Internet, but may be duplicated amongst any separate private networks. This is the basis for the concept of NAT. There can be a seemingly infinite number of networks with reserved address ranges provided they are known to the public Internet by a routable, global IP address or addresses. Figure 4 shows the standard private IP address ranges that are not globally viable:

Class A (private): 10.0.0.0 - 10.255.255.255
Class B (private): 172.16.0.0 - 172.16.255.255
Class C (private): 192.168.0.0 - 192.168.255.255

Figure 4. Private IP Address Ranges [AAPI]

2. Network Address Translation

According to RFC 2663 [IPNATTC], "The term 'Network Address Translator' means different things in different contexts." This section will cover many of the different forms and uses of NAT and will focus on basic NAT, since it will be implemented in the thesis development.

a. Basic NAT

RFC 3022 [TNAT] specifies what most people refer to when they use the term NAT. NAT was introduced as a short-term solution to the Internet address space crowding until long-term solutions with larger address spaces were accepted. Its operation depends on adherence to the private/public IP addressing scheme and the placement of NAT functionality on all network devices that form the border between the local area network using private IP address space and the Internet. The local, private addresses can be re-used by any other local area networks not directly connected to the same border device, while the global addresses are unique to the Internet. Besides the primary advantage of effectively alleviating the strain on the IP address pool, NAT also hides the local area network topology (see Figure 5) from outside hosts. According to RFC 3022, NAT also "takes advantage of the fact that a very small percentage of hosts in a stub domain [(local area network)] are communicating outside of the domain at any given time."

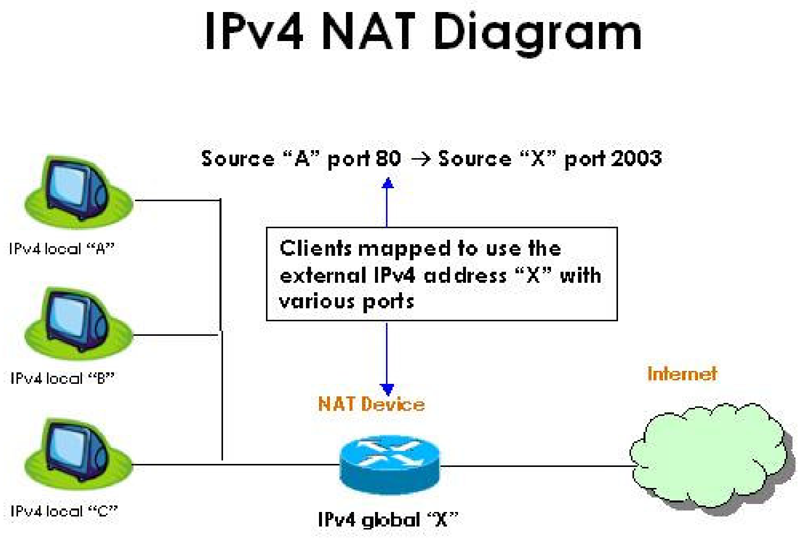

IPv4 NAT Diagram
Source "A" port 80 → Source "X" port 2003
Clients mapped to use the external IPv4 address "X" with various ports
IPv4 local "A"
IPv4 local "B"
IPv4 local "C"
NAT Device
IPv4 global "X"
Internet

Figure 5. IPv4 NAT Diagram

What follows are the core steps to a basic NAT translation, also referred to as traditional NAT or outbound NAT, which only allows connections to be initiated from the inside:

(1) Address assignment - NAT devices bind globally unique and locally re-usable IP addresses at the beginning of a network connection to the address fields of IP packets. At this point, there are two possible scenarios depending on whether the particular session is receiving a static or dynamic address assignment. In the case of static address assignment, the NAT device merely looks up the pre-determined private/public address mapping in its routing table and assigns IP addresses accordingly. In the case of dynamic address assignment, the NAT device selects a globally unique IP address from its address pool,

maps it to the local IP address and stores the connection pairing in its NAT table.

(2) Address translation and lookup – Anytime an outbound packet crosses the NAT device, the source/destination IP address pair is looked up in the NAT table to see if connection information exists. Once connection information is either found or created (for new sessions) the NAT device strips the private IP address off of the packet and replaces it with a globally unique address. Additionally, the NAT device must recalculate the IP checksums, as well as, other fields that relate to the original source/destination IP address. Incoming packets have the selected global IP address as the destination address. For these packets, the NAT device looks up the globally unique IP address in the NAT table to determine the corresponding local area network host, and forwards it with the proper header modifications. All of these address translations are intended to occur transparently to any of the hosts engaged in a session. However, RFC 2663 states that "the NAT function cannot by itself support all applications transparently and often must co-exist with application level gateways (ALGs) for this reason." [IPNATTC] Note that IPSec techniques that protect the contents of IP headers and are intended to preserve endpoint addresses of an IP packet cannot function with NAT, as NAT's primary role is to alter the IP address of an IP packet. NAT will, however, work with Virtual Private Networks (VPNs) and tunneling schemes that can tolerate the alteration of the IP address fields.

(3) Address unbinding – A NAT device may detect that communication between the local and remote hosts has halted for some given amount of time using

various heuristics. When this happens, the NAT connection expires for the corresponding address pair. The globally unique address is returned to the pool of available addresses for use with another mapping. New session pairings will have to be assigned to all new connections as they are encountered.

This basic series of events is what most people refer to when they use the term NAT, however, there are many NAT variants.

b. Network Address Port Translation (NAPT)

This NAT mechanism extends the concept of IP address translation mappings to include the transport layer ports. NAPT allows multiple sessions from multiple private hosts to be mapped onto one globally unique IP address by keeping track of the port numbers associated with the global address. Essentially the mappings contained in the NAT table are expanded to include the port number in addition to the IP address pair. This mechanism allows more unique combinations, thereby allowing multiple private hosts to access the Internet using one global IP address. For example, three different private IP hosts wish to start an HTTP session with an outside server. The NAT device would map each session to a specific IP/port pairing and store them in its NAT table. An example of a NAPT mapping using one globally unique IP address can be seen in Figure 6.

Private IP Address mapped to Globally Unique IP Address/Port Combination

192.168.0.1:80 =>	60.60.60.60:2500
192.168.0.1:23 =>	60.60.60.60:6489
192.168.0.249:80 =>	60.60.60.60:2502
10.255.255.255:1024 =>	60.60.60.60:5009

Figure 6.   NAPT Example

NAPT is a common instance of NAT that is used by many users to setup home networks using the single IP address provided by their Internet Service Provider. NAPT can also be used in conjunction with traditional NAT to further increase the amount of usable global space. For example, assume that a network has two globally unique IP addresses, by using NAPT, the network now has 2 (IP addresses) * 65535 (ports per IP address) = 131070 unique session mappings available.

c.   *Bi-directional NAT*

Also known as two-way NAT, bi-directional NAT allows sessions to be initiated from outside of the private network, as well as, from the inside. Bi-directional NAT employs a Domain Name Service – Application Level Gateway (DNS-ALG) that alters DNS packets to reflect any static or dynamic address mappings the NAT device will or has made. When an outside host wishes to initiate a session with an internal host, it sends a DNS query that ultimately reaches the internal host DNS server, which returns a DNS reply. If the internal host has either a statically mapped IP address or both a statically mapped IP address and a port enty, the DNS NAT device forwards the DNS reply. Otherwise, the DNS reply is altered by the DNS-ALG and the NAT device to reflect a dynamic mapping that the NAT device supplies as the IP address of the internal host. Since the mapping has been made, the external host may now initiate a session with the internal host via the returned address, assuming the reply occurs before the session information is purged from the routing tables.

d. Twice NAT

Twice NAT modifies both the source and destination address of an IP packet whenever it passes through the NAT device. This is necessary when a private network (improperly most of the time, sometimes on purpose) labels one or more of its internal nodes with public IP addresses officially assigned to other networks. The reasons for this address misuse vary, but the result is that a conflict arises when a host from the offending network must communicate with the public network. Because of the duplication in IP addresses, the packet is forwarded to another local host instead of the public host. Twice NAT attempts to solve this problem by altering both the source and destination address as the packet travels. Figure 7 gives a snapshot example of twice NAT.

```
Twice NAT Configuration:
Private to Public:  200.200.200.0/24 => 138.76.28.0/24
Public to Private:  200.200.200.0/24 => 172.16.1.0/24

     Datagram flow:  Private => Public
     a) Within private network
               Dest.Addr.: 172.168.1.100
               Src.Addr.:  200.200.200.1
     b) After twice-NAT translation
               Dest.Addr.:  200.200.200.100
               Src.Addr.:  138.76.28.1

     Datagram flow:  Public => Private
     a)Within public network
               Dest.Addr.:  138.76.28.1
               Src.Addr.:  200.200.200.100
     b)After twice-NAT translation, in private network
               Dest.Addr.:  200.200.200.1
               Src.Addr.:  172.16.1.100
```

Figure 7. Twice NAT Example [IPNATTC]

e. Multihomed NAT

This terminology refers to the concept of using multiple NAT border devices in a network. For NAT to be effective it must process all packets being sent to the internal network, essentially creating a single point through which all external communications must pass. Users quickly realized that this created a bottle neck in traffic, as well as a single point of failure for the network with respect to external connectivity. Multihomed NAT enables a private network to have several exits to external networks, which allows for redundancy in communications and better use of routing efficiency algorithms. This approach requires that all NAT devices maintain the same routing information. Otherwise packets will be incorrectly dropped, routed inefficiently, or have duplicated session entries in the tables. Methods for NAT information exchange vary, but all produce the same result: all NAT boxes have the same tables.

B. NAT IN THE LINUX OS

Because the chosen platform for this thesis is Linux Red Hat 9.0, it is important to understand how NAT functions within the Linux 2.6.5 kernel. This section will examine the primary packet monitoring mechanism within the kernel, *netfilter*, and both the kernel-space and user-space code of the *iptables* implementation that supports NAT.

1. Netfilter

To understand NAT within the Linux OS, it is important to have a broad picture of what happens to a packet entering a Linux system. A packet entering a network interface on a Linux computer goes through a series of "sanity checks" which include packet checksum, destination (if it is, in fact, destined for this computer), etc. in order to determine if what is received is a valid packet. Any packet failing these checks is dropped. Following these sanity checks is the first instance of a *netfilter* hook. Effectively, "*netfilter* is a set of hooks inside the Linux 2.4.x kernel's network stack, which allows kernel modules to register callback functions called every time a network packet traverses one of those hooks." [MOSIX] In essence, each hook provides an opportunity for a kernel module to look at and manipulate the packet before it continues (or is dropped) down the routing chain. This approach provides more modularity than implementing both *netfilter* and the underlying NAT code as a monolithic block of kernel code. The layering is inherent in the setup of "kernel to *netfilter* to *iptables* processing stack" and since the traversal of the *netfilter* hooks and queues is linear, any introduction of a looping problem would be the result of poorly written code. Each *netfilter* hook contains a prioritized list (it may be empty) of the kernel modules that must access the packet when the hook is activated. The *netfilter* hooks accept the following return codes from the processes, following any alterations the process may chose to do: NF_DROP (drop the packet), NF_ACCEPT (keep the packet), NF_STOLEN (keep the processor and memory resources for the packet, but the process will handle the packet so *netfilter* can forget about it), NF_QUEUE (queue the packet for userspace processing). These hooks are used by *iptables* to allow other kernel space programs the ability to view or alter a packet.

Iptables is the built-in packet manipulation mechanism that processes packets according to a set of user-defined rules. The first *netfilter* hook, following the sanity checks, is the NF_IP_PRE_ROUTING hook, during which connection tracking, packet mangling, and destination NAT occur in that order. Connection tracking looks at the destination and source address fields of the packet and records them in a table for a certain amount of time. Other programs desiring to determine what connections are active can access this information through the connection tracking mechanism. Packet mangling is essentially a sequentially traversed table of rules that are applied to packets to allow kernel space programs the ability to manipulate certain fields of a packet. For instance, one could use the mangling table to perform static NAT by instituting a rule that forwards all packets with a specific globally unique IP/port address to a specific private IP/port address. Destination NAT (DNAT) modifies the destination IP address of all incoming packets using the *nat* table to determine the proper IP and/or port mappings. Variations of DNAT include redirection (back to the incoming interface), port forwarding (multiple servers), and load sharing. There is a good excerpt from Paul Russell and Harald Welte's "*Netfilter* Hacking HOWTO" that describes what happens whenever the NAT code is called:

> "Anyway, the first thing the NAT code does is to see if the connection tracking code managed to extract a tuple and find an existing connection, by looking at the skbuff's nfct field; this tells us if it's an attempt on a new connection, or if not, which direction it is in; in the latter case, then the manipulations determined previously for that connection are done.
>
> If it was the start of a new connection, we look for a rule for that tuple, using the standard *iptables* traversal mechanism, on the `nat' table. If a rule matches, it is used to initialize the manipulations for both that direction and the reply; the connection-tracking code is told that the reply it should expect has changed. Then, it's manipulated as above.
>
> If there is no rule, a `null' binding is created: this usually does not map the packet, but exists to ensure we don't map another stream over an existing one. Sometimes, the null binding cannot be created, because we have already mapped an existing stream over it, in which case the per-protocol manipulation may try to remap it, even though it's nominally a `null' binding."

After all of this occurs at the first *netfilter* hook, including the previous connection tracking and packet mangling, the packet then enters "the routing code, which decides whether the packet is destined for another interface, or a local process. The routing code may also drop packets that are unroutable."(Russell and Welte) If the packet is an incoming packet, a second *netfilter* hook, NF_IP_LOCAL_IN is called. This hook again allows kernel modules, namely *iptables*, the ability to manipulate and examine the packet based on information obtained from rules within the filter, conntrack, and mangle tables. At this point the incoming packet is passed off to other kernel modules and is no longer under control of the *netfilter* mechanism. If a packet is forwarded to be sent out of the computer, a third *netfilter* hook called NF_IP_FORWARD is initiated which allows packet mangling and filtering, as well as any other registered processes. From here, a fourth *netfilter* hook, NF_IP_POST_ROUTING, is initialized which allows packet mangling, source NAT (SNAT), and the connection tracking mechanism to access the packet. Again, as is the case with all of the *netfilter* hooks, any kernel module can access the packet at this point if they have registered callback functions with the NF_IP_POST_ROUTING hook prior to the arrival of the packet. The only other *netfilter* hook occurs when a packet originates locally and is destined to leave the system via the local network. NF_IP_LOCAL_OUT is called which allows *conntrack*, *mangle*, DNAT, and *filter* to work on the packet. The packet is then routed and triggers the NF_IP_POST_ROUTING hook mentioned previously. At each point in the *netfilter* architecture where NAT occurs, namely the prerouting, postrouting and output hooks, the aforementioned processing steps are repeated. To summarize, NAT checks with the connection tracking mechanism to see if a connection for the particular IP address pair has existed before, and if so, applies the proper rules. If not, the *nat* table is checked for rules and if a NAT rule for that address pair exists, it is applied to the originating packets and its expected reply packets. Finally, if there is no correlated rule in the *nat* tables, the originating packet and replies are assigned a null binding to prevent multiple mappings for a single session. Figure 8 provides a graphical representation of the above proceedings.

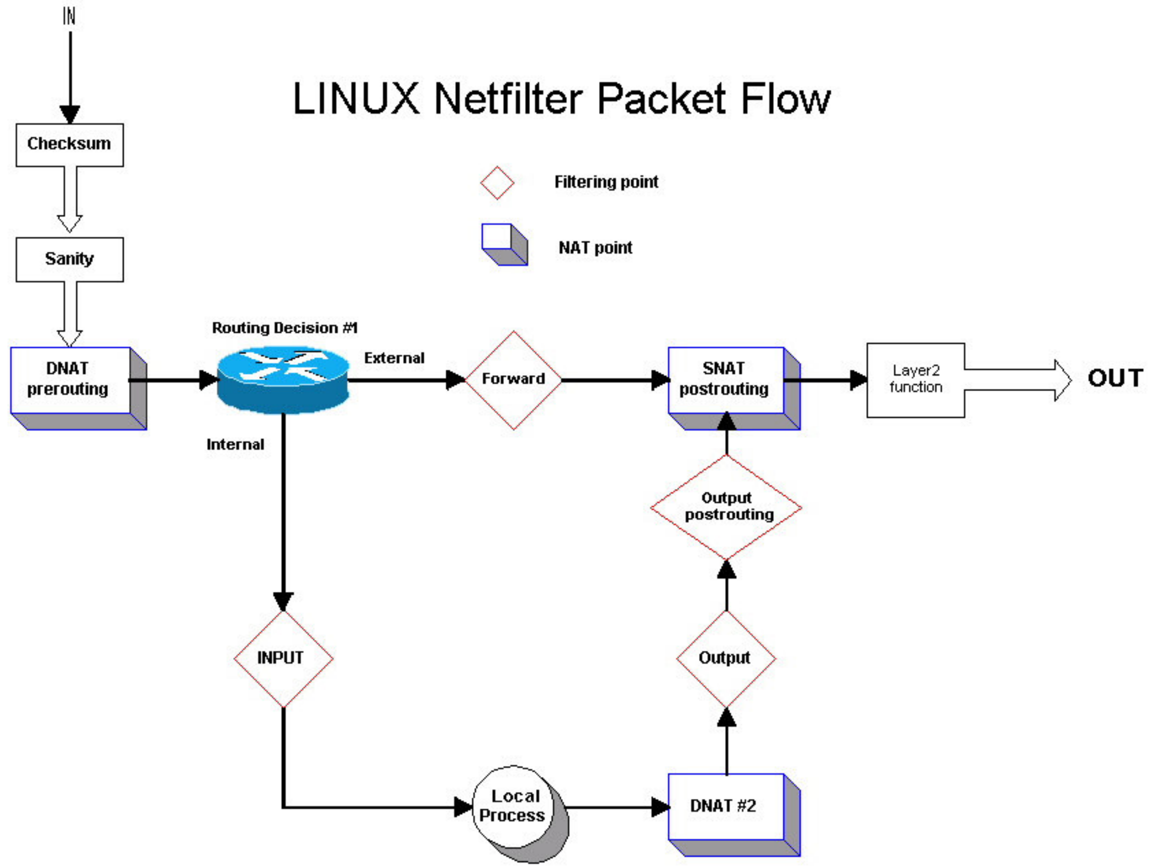


Figure 8. Netfilter Packet Flow [LNF]

2. Kernel-Space Iptables

The kernel-space *iptables* code, ip_tables.c, and related code work together to form the engine for table traversal and packet manipulation within *netfilter*. Whenever a packet reaches a *netfilter* hook, *iptables* is invoked in order to traverse each of its tables to determine if there are kernel modules that are scheduled at that hook and, if a rule exists, for that kernel module to gain control of the packet. To do this, *iptables* invokes get_entry() with the IP fields and the table to be traversed as arguments. Once a rule is returned, *iptables* then invokes other kernel modules specific to the particular table in order to perform the requested operations. For example, when a packet passes the *netfilter* pre-routing hook, destination NAT might be necessary. *Iptables* is called, with parameters including packet information from connection tracking and which table it is to traverse. At this point, *iptables* checks the nat table to see if a rule exists. If a rule does exist, *iptables* calls another function to perform destination NAT using the rule it found in the table. Should there not be any rule for the given packet, a *null-session* is assigned and nothing happens to the packet itself. These steps are repeated for all tables scheduled at the *netfilter* prerouting hook. The advantage of having a kernel-space mechanism for performing these packet manipulations is that its priority within *netfilter* for accessing the packet can be compiled into the kernel so that, if configured to do so at runtime, *iptables* can be assured the first action on an incoming packet instead of a user-space process outside of the kernel.

3. User-Space Iptables

The user-space *iptables* code supports user interation with the kernel-space *iptables* engine. *Iptables* is a command-line interface used to set various flags and rules used by kernel-space *iptables*. It then performs the requested action on the specified table. User-space *iptables* can perform any number of operations, such as listing all of the rules of a certain table, deleting a user-created table, deleting a rule within a table, appending a rule to a table, etc. It is important to note that the user-space *iptables* and the kernel space iptables share the same set of tables. Since the kernel-space *iptables* code needs to continuously access the tables in order to traverse them and apply their rules, a serious conflict would arise should the user-space process write to a table at the same time as the kernel-space process is traversing it looking for rules. A locking mechanism is employed to avoid such a conflict. When accessing the tables to perform a requested operation, the user-space *iptables* locks the tables, preventing any other process or the kernel from accessing them. Should the kernel-space *iptables* attempt to access the table while it is locked, the kernel-space *iptables* would return an NF_DROP value to *netfilter* causing the packet to be dropped so that it will have to be re-sent. Similarly, the kernel-space *iptables* would use the same lock to gain exclusive access to the tables.

C. MODULE SEQUENCE MAPPING

To better understand what happens in the *iptables* code, both kernel and user-space, it was necessary to inject tracing code into the original Linux source code and recompile. What follows is a description of what was injected and the results.

1. Kernel-Space Trace

Rather than writing completely new modules to perform a trace of what happens in the kernel space when a packet comes in, the debugging system already in place was used. Kernel debugging is essentially performed by a switch at the beginning of the code that turns debugging on or off. When debugging is turned on, various printk() statements become active and send debugging messages to the kernel logfile. The printk() statements used in this trace were customized to output the file and function that they were located in, and were put into every function of every file in the /usr/src/linux2.4/net/ipv4 directory. When implemented, this produced too many debugging statements and filled up the logfile too quickly to produce any useful results. The reason for this is that the /usr/src/linux2.4/net/ipv4 directory contains the core TCP/IP stack that is touched every time a packet enters the NIC. What was desired was the ability to trace everything that would be used within the connection tracking, *iptables*, and NAT processes. Realizing this, the kernel debugging was turned off for everything in the /usr/src/linux2.4/net/ipv4 folder except for everything in the subdirectory /netfilter. All of this was implemented on the Trusted Path Extension (TPE) computer within the MYSEA network. (More information on the network topology will be presented in Chapter 4) After successfully adding code to the kernel and recompiling it, the ICMP echo request and reply message generated by a *ping* command were successfully sent from the client machine, through the TPE, which was NAT enabled, to the server machine, and back again. NAT performed the necessary packet translations in both directions and successfully forwarded the packet.

The results from the kernel logfile indicated several things. First, that connection tracking accounted for almost all of the function calls throughout the process. Second, that ip_tables.c was only called twice during the entire process, and both times were actually during the outbound portion of the ping. The first call is to check for a specific instance of this session, which has not been established yet, and the second call is to record the session mapping that the NAT code has performed. At both points ip_tables.c calls the functions ipt_do_table() and get_entry(). Third, it appears that with return packets, NAT already knows the bindings it must perform for this specific IP mapping, and therefore does not need to call any functions from *ip_tables.c*. Fourth, if another ping packet is sent after this one, before the session mapping is unbound, *iptables.c* is only called once, because it finds the session mapping on its first try. To see a graphical representation of the logfile results from this test, and the actual logfile results, see Appendix D.

2. User-Space Trace

Another key to understanding the operation of *iptables*, in general, is to understand how the user-space *iptables* operates. In order to gain a better understanding of how this part of *iptables* works, it was again necessary to insert tracing code into the user-space source code, recompile the user-space *iptables*, and perform some basic, NAT-related commands to see what occurs. The tracing code consisted of 4 lines of code inserted into every function of every file of the user-space *iptables* code. These lines of code declared an input file, opened it, used fprintf() to send the file and function information, and closed the file. This method worked well and did not cause any compilation problems. One drawback was that it was difficult to separate the different commands, so it was necessary to manually annotate the output file after every command in order to separate the actions. The results were fairly simple. Whenever a table was manipulated via the command line interface, almost all of the functions called within *iptables* involved converting the *user-friendly* information into a more *machine-friendly* format. The process then calls functions to allocate memory space and generate the entry in proper format. Finally, depending on the exact nature of the command, its operating function is called. For example, if the NAT table was to have an entry appended onto it, after a large amount of formatting and some information processing, the actual append_entry() command is called. An important feature is that, within the final commands, just before a table is actually written to, it is locked to prevent simultaneous access by both the user-space and kernel-space *iptables*.

III. INTERNET PROTOCOL, VERSION 6

This chapter contains a summary of the IPv6 structure and functionality as it applies to this thesis. Background information regarding the protocol is presented through the analysis of relevant RFC's and supporting academic research. The application of the protocol to our thesis is explained and a comparison between NAT functionalities and those of the IPv6 protocol is examined. Finally, the current application of this protocol in the Linux kernel is explained.

A. BACKGROUND AND ANALYSIS

This section examines the history and structure of the IPv6 protocol. It also describes the IPv6 addressing scheme and address allocation.

1. Introduction

The growing demand for interconnectivity and the increasing consumer desire to have more devices wired, drove the creation of the next version of the Internet protocol. IPv6 addresses are 128-bits long. This is 4 times longer than the standardized IPv4 addresses currently in use. This is 2^{96} times the size of the IPv4 address space, allowing for hundreds of billions of additional addresses. Moreover, the most stringent studies regarding the efficiency of addressing architectures predicts that the protocol will be capable of "accommodating between 8 x 10^{17} and 2 x 10^{33} nodes" [IPng] if the IPv6 addressing architecture efficiency is comparable to that of the IPv4 addressing architecture.

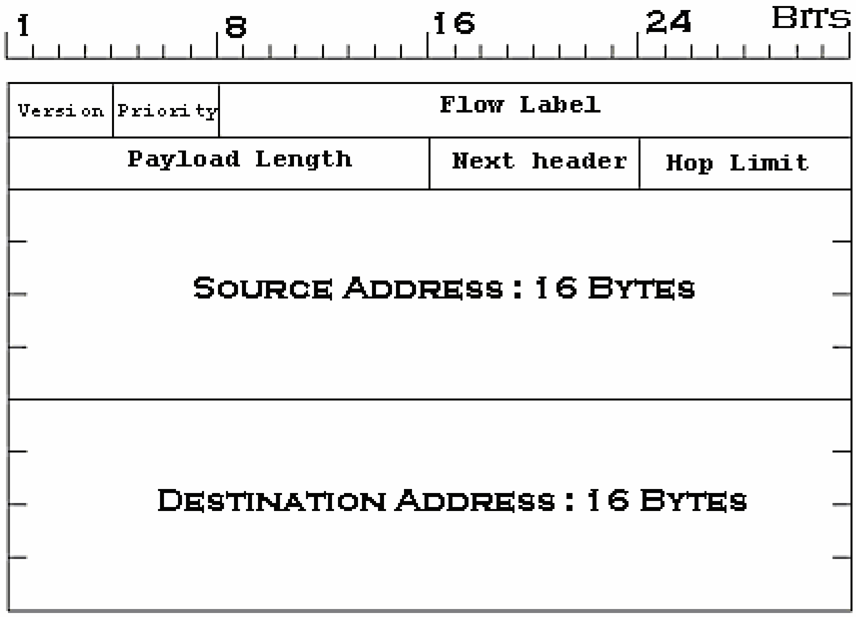


Figure 9. IPv6 Header [ACM IP6]

What follows is the bit length of each field and a description, taken verbatim from RFC 2460 [IP6 SPEC]:

- Version (4 bits) : Internet Protocol version number = 6
- Traffic Class (8 bits) : Used to identify and distinguish between different classes or priorities of IPv6 packets
- Flow Label (20 bits) : Used by a source to label sequences of packets for which it requests special handling by the IPv6 router
- Payload Length (16 bits) : Length of the IPv6 payload, i.e., the rest of the packet following the IPv6 header, in octets. (Extensions are included in this number)

- Next Header (8 bits) : Identifies the type of header immediately following the IPv6 header
- Hop Limit (8 bits) : Decremented by 1 by each node that forwards the packet. Similar to the time to live (TTL) field in IPv4
- Source Address (128 bits) : Address of the originator of the packet
- Destination Address (128 bits) : Address of the intended recipient of the packet

In the IPv6 addressing scheme, there are three different types of addresses: unicast, anycast and multicast. Noticeably, the multicast address has updated the broadcast function used in IPv4. The unicast address allows packets to be sent to an interface at a single address. This is used when the address is targeting a specific, known location. Anycast is an address that is assigned to multiple interfaces and the packet with an anycast address is sent to the most easily accessible ("closest") node. This addressing format is useful when a client needs to get a packet to the closest available server. The multicast address also identifies a set of interfaces; however, in this mode, the packet is sent to all interfaces identified by a specific address. This flexibility in addressing allows a single machine to have multiple IPv6 addresses of varying types. [IPng]

The increased address length and the number of extra nodes permitted by it present the subsequent problem of domain name resolution and address lookup. Currently, when an IP router receives a packet, the router must determine what routing subnet in its database most closely matches the incoming packet. It then routes the packet to the appropriate destination. With the address size increasing so dramatically, this method of packet forwarding quickly becomes time and memory intensive. A solution to this problem has been proposed in a collaborative effort between the University of Washington and ETH Zurich in Switzerland. This improved method creates a hash table of prefix lengths and performs a binary search. It is claimed that the search method results in an "order of magnitude performance improvement" [IPROUTE] due to only seven hash lookups needed for a 128-bit address.

2. Packet Header Format

The number of fields in the IPv6 header is greatly reduced compared to the IPv4 header, thus making it simpler and more reliable. The header in an IPv6 packet (see Figure 9) includes information about the version, the priority, the flow label, the payload length, the next header, the hop limit and the source and destination addresses. This header significantly reduces the amount of overhead that existed in IPv4 (see Figure 2), by removing the differentiated services byte, IP header length, the identification field, the flag, the fragment offset, the time to live (TTL) and the header checksum field. Removing all of these fields allows IPv6 to include a larger source and destination address without radically increasing the time spent on transmitting and receiving the header. [IPng]

Though the IPv6 header is less complex, its design has allowed for the relatively simple addition of extension headers or footers. These additional headers can serve many purposes and allow for future development of the protocol. Currently, the following headers are being used according to RFC 2460, "Internet Protocol, Version 6 (IPV6)": Hop-by-Hop Options, Routing, Fragment, Destination Options, Authentication Header (AH) and Encapsulating Security Payload (ESP). [IP6 SPEC] It is easily conceivable that future headers will provide more functionality than currently available headers and be just as easy to implement.

3. Addressing Scheme

The primary benefit of IPv6 is the increased address space. Instead of only using 32 bits in the header for a source or destination address, IPv6 uses 128 bits per address. Considering the new address space, a new address formatting scheme had to be introduced. Basically, there are three methods of expressing an IPv6 address. The first and most standard form is the following: xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx, where each x is a hexadecimal digit. Thus, an example of a typical IPv6 address is the following:

94AD:1283:BE45:9E23:FFE4:72A6:820F:7A4B.

The second method to describe IPv6 addresses is used when there are leading zeros within an octet or several of the middle octets are zero. In these cases, the leading zero can be omitted from the octet. Or, in the case of several octets being zero, the octets can be omitted and replaced with a double colon. Note that the double colon can only be used once within the address, denoting one or multiple octets that are all zero.

An example of this method is the following:

94AD:1283:0040:9E23:0009:72A6:820F:7A4B

94AD:0000:0000:0000:0000:0000:0000:7A4B

The above may be expressed as:

94AD:1283:40:9E23:9:72A6:820F:7A4B

94AD::7A4B

The following address is not valid:

94AD::ABCD::7A4B

Finally, the third method is used for addresses that are used to transition from the IPv4 protocol to the IPv6 protocol or to maintain both addresses at the same time. This format essentially allows the last two octets to represent the old IPv4 address while the first six octets represent the new IPv6 subnet.

For instance, the following IPv4 address:

192.168.100.100

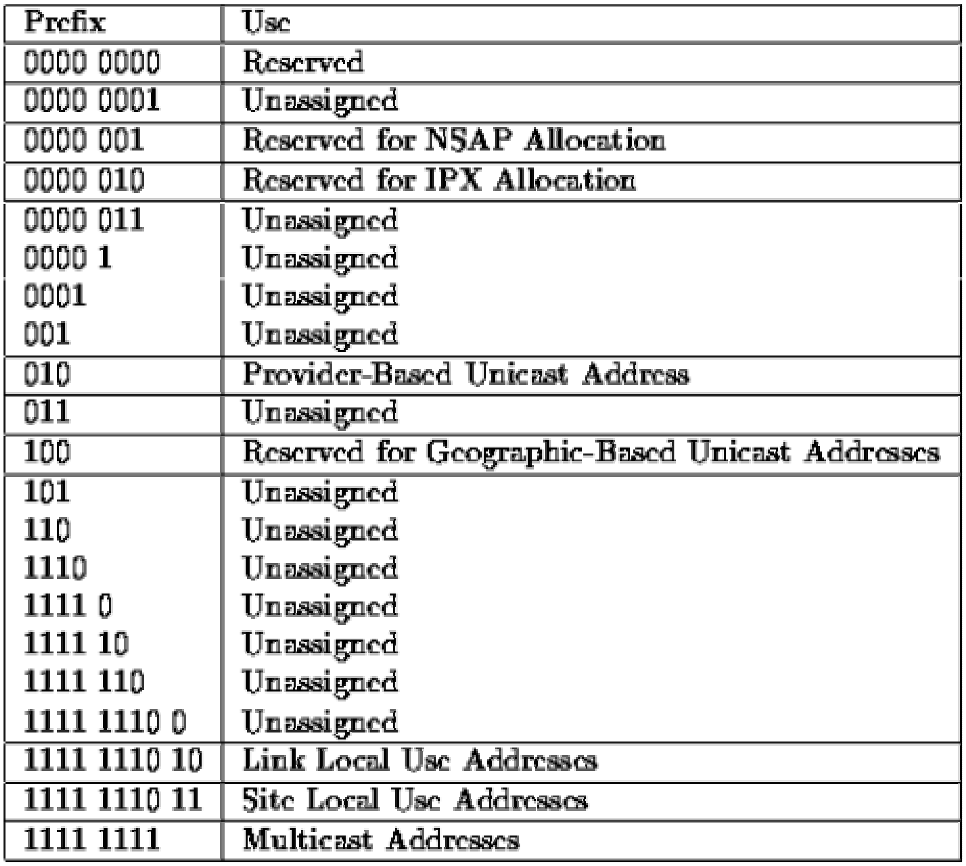
Could be translated to IPv6 with the following address:

94AD:1283:BE45:9E23:FFE4:72A6:192.168.100.100

Notice that after the sixth octet, the notation transitions from the separating colons to the current IPv4 standard of dot notation. This format in particular will be imperative in the transition from IPv4 to IPv6. Given the proper routing mechanisms, it will allow hosts to maintain sites for both protocols with relative ease.

4. Address Allocation

Just as in IPv4, IPv6 has allocated its address space through the acceptance of the initial designation presented in RFC 3513 [IP6 ADDR]. This allocation is a remarkable paradigm for future planning. As one can see in Table 1 below, the majority of addresses are unassigned and available for public use. However, a large number of addresses, proportional to IPv4, are reserved for future protocol use, link-local, site-local and multicast use.



Prefix	Use
0000 0000	Reserved
0000 0001	Unassigned
0000 001	Reserved for NSAP Allocation
0000 010	Reserved for IPX Allocation
0000 011	Unassigned
0000 1	Unassigned
0001	Unassigned
001	Unassigned
010	Provider-Based Unicast Address
011	Unassigned
100	Reserved for Geographic-Based Unicast Addresses
101	Unassigned
110	Unassigned
1110	Unassigned
1111 0	Unassigned
1111 10	Unassigned
1111 110	Unassigned
1111 1110 0	Unassigned
1111 1110 10	Link Local Use Addresses
1111 1110 11	Site Local Use Addresses
1111 1111	Multicast Addresses

Table 1. IPv6 Address Space Allocation [NGIP]

B. SECURITY

This section will examine the security functionality inherent in the IPv6 protocol as well as the additional headers available that enhance security.

1. Existing Security Mechanisms

At the time IPv4 was accepted, the security threat to packet transmission and reception was minimal. Thus, very little was built into the IPv4 architecture to protect it from threats like Man-in-the-Middle attacks, where an agent intercepts message traffic between two clients; or masquerading, where an agent masks his true identity with a false one in order to gain access to a system. IPv6 has incorporated three main deterrents for the previously mentioned attacks, an Authentication Header (AH), Encapsulating Security Payload (ESP), and Internet Key Exchange (IKE). The AH provides authentication and integrity both to the end client and forwarding server. It is able to do this by creating a cryptographic hash of the packet. If the hash is invalid upon receipt by the end client, the user knows the packet has either been tampered with or was not successfully transmitted. Using ESP, an authenticator is placed at the end of the packet. It uses the same hash mechanism as AH, however it also encrypts the payload data and the primary source and destination address. This conceals the payload of the packet to intermediate servers, giving extra security to the packet against a Man-in-the-Middle attack. This mode of packet transportation is known as tunneling mode transmission. Finally, the IKE follows the same principles as Kerberos. [KERB] A private key is encrypted with the end user’s public key, the packet is sent and then decrypted with the sender’s public key. [NGI]

2. Emerging Technologies

Packets are sent from node to node, eventually ending at the specified destination address. A router is only able to send packets to IP addresses that are stored in its routing table. Neighbor and router discovery and the sharing of routing information from those routers generate this table. A security concern arises in this situation due to the damaging potential that the unknown routers being queried might supply malicious routing information. Two emerging security methods for IPv6 neighbor and router discovery are Cryptographically Generated Addresses (CGA) and Address Based Keys (ABK). In CGA, the lower 62 bits of an IP address are used to "store a cryptographic hash of the public key." [NRD] To identify a CGA encrypted address, both bits 6, the universal/local bit, and bit 7, the individual/group bit, are set to one. The cryptographic hash of the server's public key allows the client to send an encrypted public key to be used in coordination with the server's private key, instead of sending the public key "in the clear." Using ABK, the user's private key is used to generate a digital signature and is placed in the lower 64 bits of the header. The end client then verifies that portion of the header using a public key to decrypt it. [NRD]

C. FEATURES PROVIDED BY NAT FOR IPV6

The following section is an examination of the benefits of NAT for IPv6 and how they apply to this thesis and the MYSEA architecture.

1. Address Hiding

The implementation of a network address translation protocol (NAT) for IPv6 will hide internal addresses on a private network from external view. The new IPv6 protocol does not have any methods for address hiding and cannot inherently hide internal network addresses from external view. IPv4 NAT provides address hiding as a result of its address translation. Every computer on the external side of a NAT device only sees and communicates with the NAT device. When outbound communications occur, the NAT device strips off the source headers and changes them to correlate with the pool of publicly acceptable addresses assigned to the device. When the destination computer receives the packet, it returns communications via the NAT assigned address and port, not the native private address of the computer behind the NAT device. The only way a computer outside the network could initiate communication with a computer behind a NAT device would be if the internal computer was statically bound to a particular IP/port mapping. Meaning that there is a constant 1-to-1 relationship between the true address of the client and the address of the client after being translated. Even then the NAT device still performs an address translation on all incoming and outgoing packets.

2. Dynamic Address Assignment

NAT for IPv6 will provide dynamic address assignment. IPv4 NAT provides dynamic address assignment through its address mappings and translations, unless they are statically bound. This provides an advantage because each new connection is tracked and mapped to different ports or ranges of addresses by the NAT mechanism. This makes it difficult for an outsider to determine which computer in the NAT-protected network they are communicating with. Thus, enumeration and mapping of the network are extremely difficult, which in turn makes certain forms of hacking more difficult. IPv6 does not provide any method for obscuring the network topography, nor does it provide any method for translating addresses. NAT for IPv6 will provide this functionality.

3. Transitioning Mechanism

NAT for IPv6 will help provide a transition from IPv4 to IPv6. Currently IPv4 NAT does not support a packet transitioning from an IPv4 network to an IPv6 network. However, the possibility of using an ALG or reconfiguring NAT to do so does exist. It would be relatively simple to configure NAT to encapsulate the IPv4 address into an IPv6 format in order to traverse an IPv6 network. All that would be required would be to put the IPv4 packet entirely into the data field of an appropriately labeled IPv6 packet. The NAT device would still maintain an appropriate translation table so that when the returning IPv6 packet arrives and is stripped down to IPv4 it knows which computer to forward it to. Address hiding in this way could still occur as the NAT device could strip the private IPv4 network address, assign a global IPv4 address and then encapsulate that datagram into an IPv6 packet. IPv6 does not have an inherent method of communicating with an IPv4 network without using either encapsulation or reformatting the header. This is due to the differences in structure between the IPv4 and IPv6 header formats. Again, while NAT does not support a packet transitioning from an IPv6 network to an IPv4 network, it is possible to use an ALG or to reconfigure the NAT device. In the case of the latter, either the datagram is encapsulated, or the IPv6 header is stripped off and replaced with an IPv4 header.

4. Tunneling

NAT for IPv6 will be able to tunnel end to end. IPv4 NAT is transparent to end-to-end tunneling, as stated in

RFC 2663: "All variations of address translations discussed in the previous section can be applicable to direct connected links as well as tunnels and virtual private networks (VPNs). Note also that end-to-end ESP based transport mode authentication and confidentiality are permissible for packets such as ICMP, whose IP payload content is unaffected by the outer IP header translation." [IPNATTC] IPv6 is also compatible with any application layer end-to-end tunneling as it is merely an IP layer protocol.

NAT for IPv6 will also work with link encryption. Since NAT for IPv6 is based on NAT for IPv4, it will have the same IPv4 NAT characteristics including the ability to encrypt the payload on an end-to-end basis. The NAT mechanism does not alter any payload data during transmission or reception, thus any encryption mechanisms at the application level remain untouched by NAT as does all application level data.

5. Connection Limiting

NAT for IPv6 can be used to limit the number of connections to an external network. IPv6 has no inherent method of limiting external connections or performing bandwidth shaping. However, IPv4 NAT can indirectly limit the number of external connections a network can make. By limiting either the pool of IP addresses from which a NAT device can assign translations, or the range of ports that can be assigned for port translation, a NAT device can effectively control traffic flow. For example, suppose one wishes to limit the number of external connections from a network to 50. By limiting the NAT device to a 50-port range, all additional requests would result either in the packet being dropped or in an error message. This provides a unique way to shape the bandwidth of a network, and can possibly act as a security measure by preventing the NAT network from becoming a participant in some sort of zombie or DoS attack.

D. DESIRED NAT FEATURES NOT PROVIDED BY IPV6

Mapping out the respective features of IPv6 and NAT individually allows a comparison to be drawn between the two. From this comparison, the desired NAT functionality that IPv6 does not provide can be discerned with the ultimate goal of understanding the benefits IPv6 NAT has over just IPv6. The primary benefit that NAT provides to a networked computer, one neither IPv4 or IPv6 provides, is address hiding. By altering the incoming and outgoing addresses of IPv4 or IPv6 packets, the true IP address of an internal computer cannot be seen by an outside computer. IPv6 has no inherent mechanism for this, while NAT provides this implicitly as part of its implementation. The only time an internal computer can be externally identified is when there is a static NAT mapping to an external address. This ability to hide the topography of a network provides an additional layer of security by disrupting a hacker’s attempt to enumerate the network. Another additional benefit of NAT is that it prevents broadcasts from traversing it, thereby preventing broadcast attacks. Additionally, since only internally initiated connections will be allowed through dynamic NAT (static NAT is used for externally initiated sessions through the NAT device) any attempts to flood a network would be stopped at the NAT device. Besides internal network security, NAT can also be used to limit the number of simultaneous connections by limiting the pool of mappable addresses.

Ultimately, the primary benefit of NAT that IPv6 does not provide is the ability to mask the internal network from the external viewer. This is well stated by RFC 3022, titled the IP Network Address Translator: “On the other hand, NAT itself can be seen as providing a kind of privacy mechanism. This comes from the fact that machines on the backbone cannot monitor which hosts are sending and receiving traffic (assuming of course that the application data is encrypted).”[TNAT] Most other security benefits from NAT are derived from the primary benefit of address hiding. Even though there were not many other discernable benefits of implementing NAT in IPv6, address hiding alone is enough to merit the addition of NAT to IPv6.

E. IPV6 SUPPORT WITHIN THE LINUX KERNEL

This section will examine the current IPv6 functionality within the Linux kernel (version 2.6.5) that will be used in the remainder of this thesis. It is vital that the current functionality that supports IPv6 within the kernel is understood so that the existing functionality is not used effectively and is not duplicated.

1. Initialization

In the Linux kernel version used for this thesis, as in all current kernel releases, IPv6 protocol support is available as a loadable kernel module, and is not pre-loaded by default. To enable IPv6, the developer can either load the module with the command “modprobe ipv6” or set the IPv6 module initialization switch to “yes” in the *ifcfg* file for each interface. If the development is an ongoing process, the latter of the two options will be more efficient for the developer. Once the module is loaded, the Ethernet interface, unless the IPv4 module is turned off, will act in dual-stack mode. This allows the interface to receive both IPv4 and IPv6 packets. Loading the module will also assign each interface controlled by the kernel an IPv6 address. It will be a link-local address that is based on the interface's MAC address. The system now has connectivity to IPv6 devices connected directly to it.

2. User-Space Functionality

A large part of the *iptables* command line interface for IPv4 functionality has been directly adapted for IPv6 usage. Consequently, much of the formatting is exactly the same, except for a few v6 notations. Examples of these are the *ping* function and the *traceroute* function. The switches are predominantly the same, however the syntax is *ping6 <IPv6 address>*. This formatting is somewhat consistent throughout the multiple user interfaces. For example, with the Netfilter *ip6tables*, the syntax is nearly the same as the syntax for IPv4 *iptables* regarding the switches and inputs.

Unfortunately, there are several functionalities missing in IPv6 that were present in the IPv4 protocol. Namely, the *nat* table within *iptables* is not present within *ip6tables*. The primary reason for this is due to the developers' lack of priority for developing connection tracking for IPv6 within the Linux kernel. Since there is such a large address space in IPv6, it was thought that the network address translation functionality would not need to be ported from IPv4. [NONAT] Also, since there is no connection tracking, some of the filtering rules through *ip6tables* do not work, such as the filtering based on TCP sequence number tracking.

3. Kernel-Space Functionality

Much of the kernel-space *ip6tables* and *netfilter* IPv6 functionality is directly adapted from the current IPv4 functions. There are several functions within the IPv6 portion of the kernel that even state in the source code that they are blatant copies of the IPv4 source code with function name changes and different header files. When the source code and file structure of the two protocols is compared, it is obvious that functionality is basically being duplicated and syntactically manipulated to work with a different header structure. (See IPv6 Module Sequence Mapping & Directory Comparison Appendices) It is debatable whether this is beneficial or not to the Linux and *netfilter* communities. It could be argued that since the code and functional structure worked in the IPv4 environment, it is not necessary to change it for IPv6. Conversely, if the programming community at large allows a blind direct port, it is possible that the port could adversely impair the modularity of future enhancements to the code.

The source code and file structure within the kernel-space *iptables* and *netfilter* supporting IPv4 and IPv6 are somewhat similar. As stated previously, many of the functions are direct copies of IPv4 functions adapted to work with the IPv6 protocol. Much of the functionality however, was grouped differently with regards to the file system. For the most part however, the resulting function calls from a given networking action produce relatively similar output.

IV. MONTEREY SECURITY ARCHITECTURE

This chapter contains a summary of the Monterey Security Architecture (MYSEA): both its purpose and its topology as they relate to this thesis. The idea and design of which originated from the problem of achieving multilevel security in a high assurance manner. By enforcing mandatory security policies, this architecture can support such government and military contexts as coalition environments, inter-Department dependencies created by the Homeland Security Department and the Global War on Terrorism.

A. INTRODUCTION

The basis of the MYSEA project is to provide "a trusted distributed operating environment for enforcing multilevel security policies." [MYSEA] The MYSEA architecture provides centralized management while maintaining compatibility with existing consumer applications. MYSEA is a heterogeneous architecture that consists of low-assurance, commercial off-the-shelf (COTS) clients, specialized authentication devices, and a small number of MLS servers (see Figure 10). High assurance capabilities are achieved through the policy enforcement by a high assurance platform, namely the DigitalNet XTS-400 which supports the high assurance labeling of subjects, objects and networks. [MYSEA] MYSEA allows an organization to implement high assurance security without the need to completely replace their existing network. The only additional hardware needed would be the MLS server and a set of specialized authentication devices, such as the Trusted Path Extension (TPE).

B. ARCHITECTURE

The MYSEA design is primarily a two-tier, client-server relationship. The client, through the Trusted Path Extension (TPE), authenticates itself at a given session level with the MYSEA MLS server. The client is then recognized by the server for the remainder of the session at the authenticated level and is therefore authorized for information of that classification. It may seem that since the client must authenticate through the TPE, that the architecture is in fact a 3-tier architecture, similar to that found in most web-based database clients. Though the client must authenticate itself through the TPE, the two entities can actually be viewed as one node to the server and to the outside network. The TPE, acting as an extension of the MLS server, provides a trusted path for the user to authenticate with the server.

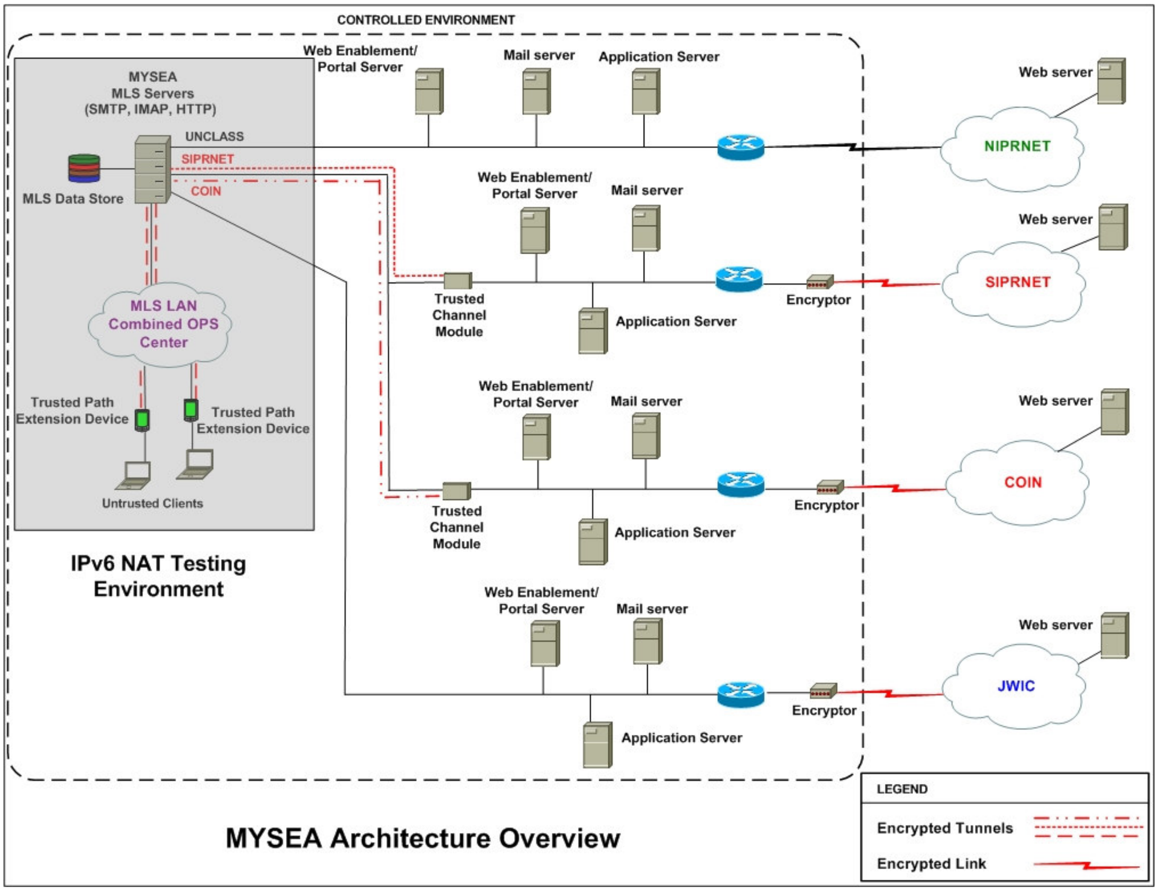


Figure 10. MYSEA Architecture [MYSEA]

In the MYSEA framework, a large majority of an organization’s network can stay relatively the same. The primary differences in the MYSEA architecture as compared to the standard enterprise architecture are the TPE, the MLS servers, the Trusted Channel Modules (TCMs) and the border data link encryptors. As noted by the diagram, the TCM and the border encryptors are outside the scope of this thesis. The TCM though, provides basically the same functionality as the TPE, however the TCM authenticates a data link to the server whereas the TPE authenticates a client. [MYSEA]

C. IPV6 NAT TESTBED COMPONENTS

Only the MLS LAN portion depicted in Figure 10 is within the scope of this thesis and is used as the basis for the IPv6 NAT testbed. Abstractly, the network sees the TPE and the client as one device. The TPE must perform NAT to hide the address of the client. The following diagram (Figure 11) shows the IPv6 NAT testbed on which the analysis, development, testing and implementation of this thesis occurred.

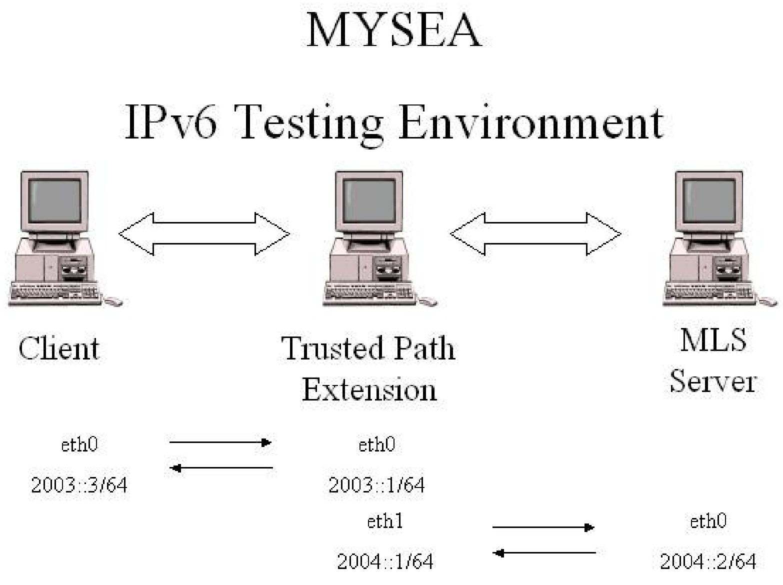


Figure 11. MYSEA IPv6 NAT Testbed

As illustrated, the testbed topology consists of two subnets, the 2003 subnet and the 2004. The TPE within the IPv6 NAT testbed is equipped with two Ethernet cards and is consequently able to forward packets between the client and the MLS server. A simple addressing scheme was used for ease of maintenance within the IPv6 NAT testbed. This scheme assigns the middle six octets in the address as well as the first three digits in the last octet to zero. The following is a description of each component within the IPv6 NAT testbed.

1. MYSEA Server

In the MYSEA architecture, the server runs the DigitalNet STOP operating system on top of the XTS-400 platform. [MYSEA] The reason for this is to make the best use of the Bell and LaPadula as well as Biba policies supported by the system. For this thesis however, the server is an earlier prototype of the MYSEA server that runs on a modified version of OpenBSD 3.1. The modified OS has the ability to label data at different classifications. It does not have the required level of assurance for a MYSEA server. There are relatively few of the DigitalNet servers that will be used in the MYSEA architecture and are fairly expensive, thus development and testing on other, less cost prohibitive equipment was acceptable. Also, since the protocols that will be used for NAT will be same regardless of the server or client systems. [MYSEA COMP]

2. MYSEA Trusted Path Extension

The TPE is an extension of the MLS server, providing an unforgeable interface for the user to authenticate with the MLS server. The principal importance of the TPE is that it is non-bypassable by the client. All traffic that is transmitted or received by the client must first pass through the TPE. This is a mechanism that cannot be subverted, regardless of the sophistication of the malicious software.

The TPE can take the form of a separate device from the client CPU. Herein, all network traffic leaving the Ethernet device must first pass through the TPE before reaching the server. The TPE can also take the form of a specially designed Ethernet card with a separate processor and memory. The TPE can possibly even be a cutting-edge Common Access Card (CAC) with its own processor. The primary concern is that the TPE's domain is separate from the client's with regards to the processor and memory. This ensures that the trusted path will not be corrupted by malicious activities that might take place on the client

In the IPv6 NAT testbed environment, the TPE is a PC with a heavily modified version of Linux 2.6.5 running on an Intel x86 processor. As noted previously, the TPE maintains two separate Ethernet cards on two different subnets. Currently, the TPE in the IPv6 NAT testbed environment does not run the actual TPE code. It only emulates the NAT functionality of a TPE and maintains the non-bypassability characteristic inherent in any trusted path. The packets transmitted or received by the client must first be forwarded by the TPE before reaching the server.

It is for this reason that the NAT functionality is placed within the TPE. Therefore, as a result of the architecture, one could hide either a single client or an entire network of clients behind the TPE. From the viewpoint of the MLS server, the network topology appears as if the server is only in communication with the TPE. In reality, there could be one or more nodes hidden behind it. Though it is possible to conceal multiple systems behind one NAT device, the MYSEA architecture is designed for one TPE for every client. The goal of this thesis is to achieve this concealment for the MYSEA client in an IPv6 environment.

3. MYSEA Client

The MYSEA client is intended to be a diskless COTS system running unmodified end-user applications. This client will have enough RAM to run various applications at the same time. Client memory will be reset when a session is terminated and all user-specific files and settings will be stored on the MLS server.

In the IPv6 NAT testbed, however, the client is currently a Linux 2.4.20-8 kernel running on an Intel x86 processor. This was done primarily to facilitate the testing of NAT in an IPv6 environment. Since Linux has built-in support for IPv6 and *netfilter*, it was chosen for developmental reasons. The most important functionality requirement for the client system in the testbed environment is the capability to test multiple network protocols over the NAT environment as well as monitor the network traffic from its Ethernet interface.

THIS PAGE INTENTIONALLY LEFT BLANK

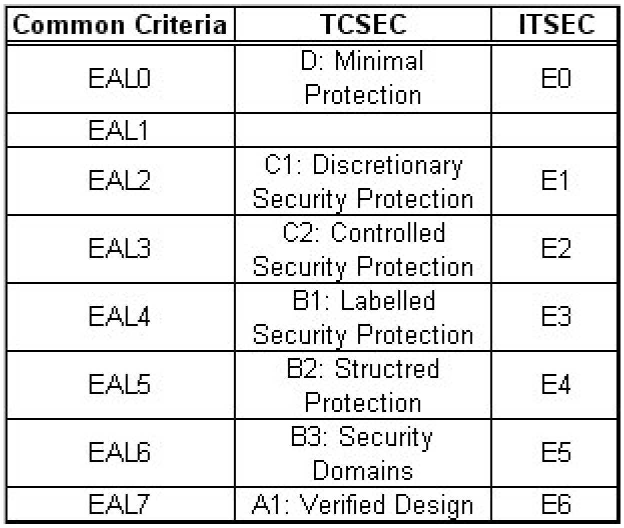
V. COMMON CRITERIA ASSURANCE LEVEL EXPLORATION

This chapter contains a summary of the Common Criteria security evaluation process, a presentation of requirements for Evaluation Assurance Level 5 (EAL5) and a discussion of how some of these requirements were used to guide the IPv6 NAT implementation. The IPv6 NAT mechanism implemented for this thesis is primarily a one-to-one port of the existing IPv4 NAT mechanism, thus the implementation does not satisfy many of the EAL5 requirements.

A. COMMON CRITERIA BACKGROUND

The Common Criteria (CC) was created as a solution to the multiple international standards that were intended to independently regulate the field of IT security evaluation. Before 1999, when the CC was adopted as an ISO standard, several standards existed including the Information Technology Security Evaluation Criteria (ITSEC) of Europe, the Trusted Computer System Evaluation Criteria (TCSEC – Orange Book) of the US and the Canadian Trusted Computer Product Evaluation Criteria (CTCPEC) from Canada. A comparison of the assurance evaluation levels between the aforementioned standards can be found in Table 2.

The intent of the CC was to create a standard set of components that define the security requirements needed to categorize IT products by assurance and functionality. The CC provides a great deal of flexibility in that the design team for a particular IT product can specify the security functionality within the definition of the protection profile for that class of IT products if one exists. The design team also has the flexibility to select the assurance level at which the product is evaluated.



Common Criteria	TCSEC	ITSEC
EAL0	D: Minimal Protection	E0
EAL1		
EAL2	C1: Discretionary Security Protection	E1
EAL3	C2: Controlled Security Protection	E2
EAL4	B1: Labelled Security Protection	E3
EAL5	B2: Structred Protection	E4
EAL6	B3: Security Domains	E5
EAL7	A1: Verified Design	E6

Table 2. Assurance Evaluation Comparison [CC WWC]

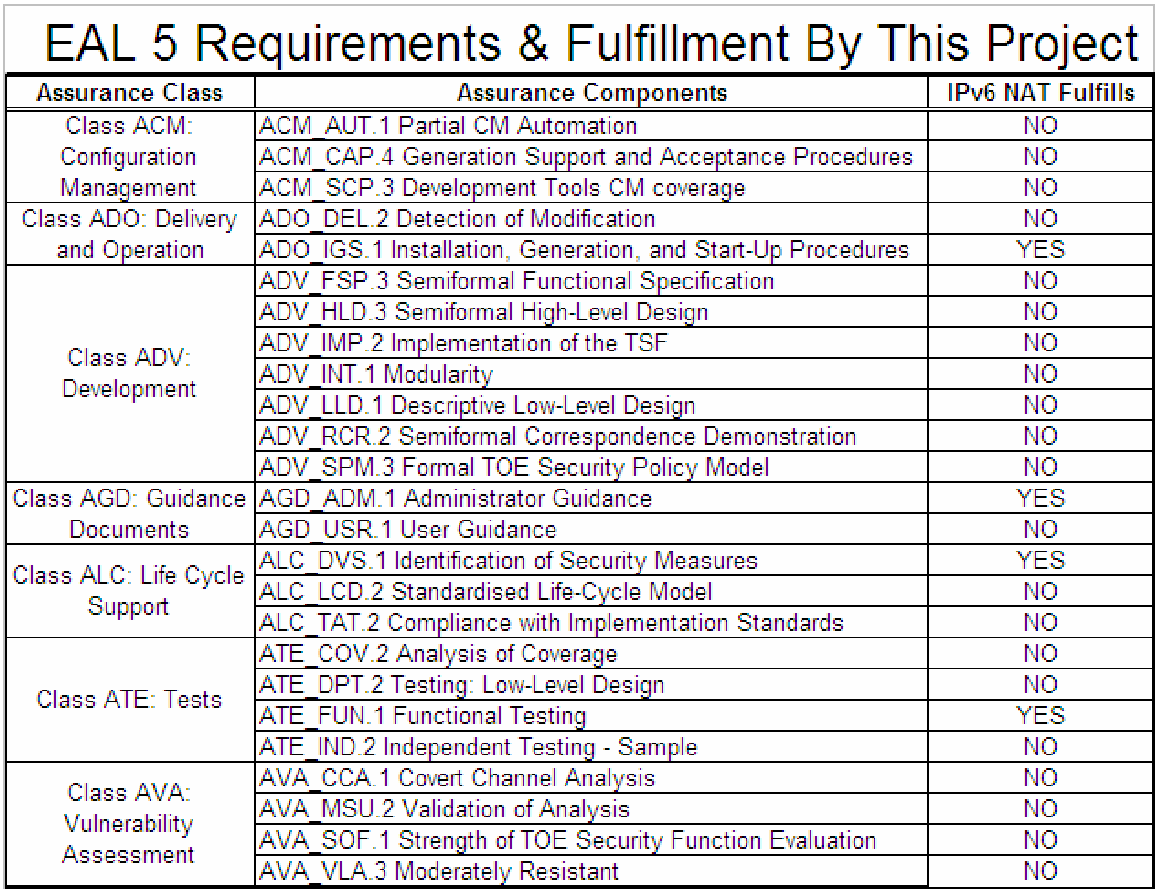
B. EVALUATION PROCESS

The CC format requires developers to have their IT product or code independently evaluated by a third-party using a common set of evaluation standards. This process involves the examination of the IT product for claimed functionality as well as for adherence to a stated set of security requirements. This evaluation is performed by an independent testing lab and can be costly in terms of both time and money. The major benefit received from this evaluation is the ability to give confidence in the product to the end-user based on a guaranteed security assurance level.

For a CC evaluation there are two principal components that the developer must provide to the independent evaluator: the Target of Evaluation (TOE) and the Security Target (ST). The Protection Profile (PP) is optional, however it can provide a more abstract statement of security objectives to which many security targets may be conformant.

The Protection Profile (PP) defines the set of security objectives and requirements (both functional and assurance) for an IT product class. Product categories include but are not limited to: firewalls, intrusion detection systems (IDS), key recovery, operating systems (OS), peripheral switches and tokens. These are the categories for which, at the time of this publication, a valid US Government PP exists. If the IT product claims conformance to a PP, then the validation and fulfillment of the appropriate profile is required for certification of the IT product. [CC SECEVAL]

The Security Target (ST) contains the security objectives and requirements for a particular IT product. The level to which the independent lab examines the TOE's assurance measures and functionality is dependent on the desired Evaluation Assurance Level (EAL). As illustrated in Table 2, the EALs correlate to the evaluation levels of TCSEC and ITSEC, with EAL7 being the highest evaluation level and EAL0 the lowest.



EAL 5 Requirements & Fulfillment By This Project

Assurance Class	Assurance Components	IPv6 NAT Fulfills
Class ACM: Configuration Management	ACM_AUT.1 Partial CM Automation	NO
	ACM_CAP.4 Generation Support and Acceptance Procedures	NO
	ACM_SCP.3 Development Tools CM coverage	NO
Class ADO: Delivery and Operation	ADO_DEL.2 Detection of Modification	NO
	ADO_IGS.1 Installation, Generation, and Start-Up Procedures	YES
Class ADV: Development	ADV_FSP.3 Semiformal Functional Specification	NO
	ADV_HLD.3 Semiformal High-Level Design	NO
	ADV_IMP.2 Implementation of the TSF	NO
	ADV_INT.1 Modularity	NO
	ADV_LLD.1 Descriptive Low-Level Design	NO
	ADV_RCR.2 Semiformal Correspondence Demonstration	NO
	ADV_SPM.3 Formal TOE Security Policy Model	NO
Class AGD: Guidance Documents	AGD_ADM.1 Administrator Guidance	YES
	AGD_USR.1 User Guidance	NO
Class ALC: Life Cycle Support	ALC_DVS.1 Identification of Security Measures	YES
	ALC_LCD.2 Standardised Life-Cycle Model	NO
	ALC_TAT.2 Compliance with Implementation Standards	NO
Class ATE: Tests	ATE_COV.2 Analysis of Coverage	NO
	ATE_DPT.2 Testing: Low-Level Design	NO
	ATE_FUN.1 Functional Testing	YES
	ATE_IND.2 Independent Testing - Sample	NO
Class AVA: Vulnerability Assessment	AVA_CCA.1 Covert Channel Analysis	NO
	AVA_MSU.2 Validation of Analysis	NO
	AVA_SOF.1 Strength of TOE Security Function Evaluation	NO
	AVA_VLA.3 Moderately Resistant	NO

Table 3. EAL5 Fulfilled Requirements By This Project [CC]

The Target of Evaluation (TOE) is the actual IT product that is to be evaluated by the third-party lab. The PP defines the scope of the product for the specific category of evaluation that the TOE must satisfy in order to claim conformance. [CC SECEVAL]

C. EAL5 REQUIREMENTS

For an IT product to receive an EAL5 certification, it must satisfy a series of conditions that verify its evaluated level of assurance. The security assurance evaluation of an IT product includes verifying its configuration management (CM), delivery and operation, development, guidance documents, life cycle support, testing, and vulnerability assessment. A summary of these requirements can be found in Appendix F. Table 3 illustrates the fulfilled assurance components by this implementation for an evaluation level of EAL5. [CC SECEVAL] The following sections describe the security assurance requirements that were partially satisfied by this NAT for IPv6 implementation.

1. Installation, Generation and Start-Up

The installation guide to install and setup the modified kernel with NAT functionality for IPv6 is described in Appendix G. This guide patially satisfies the ADO_IGS.1 requirements described in Appendix F, Section 5.

2. Administrator Guidance

Appendix E describes how to administer and use the NAT mechanism for IPv6 developed for this thesis. This guidance manual is intended for use as the *man page* for the *ip6tables* service provided by the Linux kernel. This manual partially satisfies the AGD_ADM.1 requirements described in Appendix F, Section 13.

3. Development Security

This development meets these requirements on many levels. First, the development occurred at the Naval Postgraduate School, which is currently subject to the Department of the Navy Force Protection measures. Second, a cipher-locked door that remains shut at all times protects the lab in which development occurred. Finally, the computers used for development are protected by identification and authentication mechanisms that validate the identity of the user to prevent unauthorized access on the development system. Since there was no prior written plan or procedures regarding security, these measure only partially satisfy the ALC_DVS.1 requirements described in Appendix F, Section 15.

4. Functional Tests

Appendix D provides the results from testing the functionality of the IPv6 NAT implementation. This requirement is only partially fulfilled since there are no test plans, procedures or documentation. These test results partially satisfy the ATE_FUN.1 requirements described in Appendix F, Section 20.

VI. DEVELOPMENT AND TESTING OF NAT FOR IPV6

This chapter discusses the development and implementation of NAT for IPv6, which primarily is a one-to-one port of the IPv4 *netfilter* NAT mechanism. The order in which the layer 3 and layer 4 checksums were calculated had to be reversed because the standard IPv6 header structure does not containing a checksum. Additionally, the introduction of a pseudo-header checksum to ICMPv6 required the functionality to be restructured. These modifications, as well as, porting methodology, testing procedures, and debugging outputs will also be discussed. In addition, the specification document for this project can be found in Appendix B.

A. CONNECTION TRACKING

For NAT to function properly, it must be able to track connection information for each initiated session. This allows NAT to translate a packet to the proper internal IP address. Otherwise, NAT would not be able to determine if an incoming packet is attempting to initiate a new session, or if it is a reply to a previously established connection. For IPv4, the *netfilter* connection tracking module performed this function by capturing and storing session information accessable to any number of processes to access, NAT being one of them. However, from the time IPv6 was integrated into the kernel up to the latest standard 2.6.5 Kernel distribution, connection tracking for IPv6 was not developed.

The Universal Playground for IPv6 (USAGI) project develops and distributes IPv6 programs and kernel patches for interested developers. Included in some of the more recent Linux 2.6 kernels is a connection tracking module for IPv6 that closely mirrors the IPv4 connection tracking implementation. For this thesis the IPv6 NAT mechanism was ported to run on the USAGI-altered 2.6.5 Linux kernel. Using the USAGI kernel allowed the development to focus on NAT rather than supporting functionality. This helped to shorten the development time.

B. PORTING METHODOLOGY

Since the *netfilter*, connection tracking, and user-space *iptables* framework for IPv6 had already been ported to IPv6, it was decided that a one-to-one port of the IPv6 NAT code would be the easiest way to create a functional implementation. The IPv6 programming convention used by the *netfilter* developers was maintained in the ported code. This port was easier because it was done on the same hardware architecture, Intel X86, instead of crossing over to some other hardware architecture, SPARC for example. Additionally, porting this code using the same operating system made the process easier. Porting between Linux and Windows would have been far more difficult than to and from Linux.

Performing this port using the same hardware and operating system prevented some potential difficulties. For example, recompiling the Linux kernel requires complex configuration for hardware dependancies, but using the same hardware allowed the same configuration to be used each time. A majority of the one-to-one port involved copying the existing IPv4 NAT code into the IPv6 codebase, and then changing variable names, function names, and references to reflect IPv6 values. Problems other than simple porting errors are discussed later. Using the same porting methodology as the *netfilter* developers allowed the coding process to transition much more smoothly than if an entire restructuring of the code had been attempted.

1. User-Space Iptables

Iptables package 1.2.9 contained the latest user-space *iptables* and *ip6tables* implementation available at the time of the IPv6 NAT development. Since NAT was not ported to the IPv6 *ip6tables*, this version did not have the logic necessary to interact with the kernel-space IPv6 NAT code.

In order to have basic NAT functionality, it was necessary to perform a one-to-one port of the source NAT (SNAT) target. SNAT provides the logic needed to allow the user-space *ip6tables* to interact with the nat table shared by the kernel-space. Instead of creating an SNAT target, the IPv4 SNAT target was ported to *ip6tables*. The one-to-one port was chosen because it followed the methodology of the *netfilter* programmers. Additionally configuration of the main *ip6tables* file was necessary so that it recognized the new SNAT target. This process succeeded with little difficulty as the one-to-one port was relatively straightforward.

One coding issue relative to the parsing of a port number from a given IP address range was encountered. The standard convention for designating a layer 4 port with an IPv4 address is to use a colon to separate the IP address and port pairing. For example, 192.168.100.100:80 would specify that the 192.168.100.100 IP address was to operate on port 80. This IPv4 convention does not work with IPv6, as colons are used to separate the IPv6 octets. However, RFC 2732 [IP6 URL] establishes the convention of placing brackets around the IP address to delineate it from the port pairing. For example, specifying that IPv6 address 2003::5 operates on port 80 results in the expression: [2003::5]:80. However, because this IPv6 NAT development only deals with Basic NAT, which does not deal with ports, this problem was not addressed.

2. Connection Tracking and Netfilter

Although the USAGI developers had ported the connection tracking modules for IPv6, they deliberately left out NAT-specific code. The connection tracking core source code file ip6_conntrack_core.c was modified to include NAT functionality that had not been ported from its IPv4 counterpart. This code is referenced in Appendix C.

Additionally, the connection tracking header file was changed to allow programmers to utilize NAT helpers and helper private information in connection tracking. Nat helpers are functions that help NAT process packets from applications that require more than simple layer 3 and layer 4 alteration. For example, FTP, TFTP, and AH need NAT helper functions to deal with IP information within the payload.

Unexpectedly, modifications to *netfilter* core code had to be made as well. The netfilter.c in Ipv4 file contains a function called skb_ip_make_writable that allows NAT to write the translated IP information to the networking packet buffer (skb). This function did not exist for IPv6 and had to be ported in order for NAT to change the packet in the network buffer.

3. NAT Code

The one-to-one port of the NAT code is described here. The porting process began by comparing the IPv4 *netfilter* files to those in IPv6 to determine where they differed. Each file was examined to determine if it had any relevance to NAT either as a core code file or as a supporting file. Only supporting files that are germane to this thesis (i.e., to support SNAT) were ported. Protocol-specific NAT helper files and functions such as the FTP and TFTP modules were not ported. These modules allow NAT to deal with applications needing special translation. Porting of these modules is outside the scope of this thesis.

The Change Control Procedures Appendix (see Appendix A) contains a list of all NAT files that were either modified or ported, and a brief description of their functionality.

Porting the NAT code involved updating the IPv4 code to handle the differences in format between the IPv4 and IPv6 headers. For example, since the IPv4 header can include any number of options, its size is dynamic and the NAT code has to calculate the actual header length whenever it needs to know the IP header length. However, IPv6 headers are static in length and the NAT code can use a constant index value to determine the header length of an IPv6 packet.

Another difference between the IPv4 and IPv6 code that had to be fixed was how different pointers reference specific structures and fields. For example, the IP header pointer of the network packet buffer structure (skb) in IPv4 is named iphdr, yet in IPv6 the IP header pointer is named ipv6hdr. Many of the variable names had to be altered to reflect name changes made between IPv4 and IPv6 because of the convention that already existed when *netfilter* was ported to IPv6. In addition to changing variable names, it was also necessary to edit many of the included header files and many of the included, as well as referenced, function names. For instance, the NAT code needed the ipv6.h file as opposed to the ip.h file, and it had to rename its reference to ip_conntrack_tuple to ip6_conntrack_tuple. All of these changes were based on the porting conventions used by the *netfilter* programmers when they ported the networking suite from IPv4 to IPv6.

C. PORTING DIFFICULTIES

The one-to-one method of porting NAT code provided a streamlined framework for modifying and creating code. However, several difficulties arose during development. These are described in this section.

1. IPv6 Address Structure

The first major obstacle experienced in the NAT code port dealt with differences in how the kernel handled the IP addresses. In the Linux kernel an IPv4 address is defined as an unsigned 32-bit integer, yet an IPv6 address is defined as a 128-bit structure. This structure contains a union of three arrays of 4, 8, and 16 elements, allowing the 128-bit IPv6 address to be accessed in three different formats. Many of the functions in IPv4, both NAT and other *netfilter* functions, manipulate the IP address through binary operators. However, binary operators cannot be applied to a structure, making it difficult to compare IP addresses for equality, increment or decrement IP addresses, or to perform bitwise manipulations. A simple assignment of a translated IP address to a source IP address (src.ip = nat.ip) cannot be performed with a structure data type. Some functions use bitwise manipulation as a shortcut to performing standard mathematical operations. For example, computing a checksum based only on what has changed would be a shortcut to re-computing the entire checksum after one or more fields have changed.

In most cases the solution to this problem involved accessing the IP address one array entry at a time or by assigning pointers to the array. For instance, if the above IP address assignment were to be performed through array access, the result would look something like the following: src.ip.s6_addr32[0] = nat.ip.s6_addr32[0]. The index is then incremented on each array element until it reaches 3, thereby assigning each 32-bit portion of the temporary IP address to the respective 32-bit portion of the source IP address. If the operation were to be done with pointers, the contents of the location pointed to by the source IP pointer would be assigned to contain the contents of the location pointed to by the temporary IP pointer. In some cases the solution simply involved using existing functions already ported by *netfilter* and USAGI programmers.

2. Checksum Calculation Ordering

When the NAT mechanism alters the header of a packet, it must recalculate the packet's checksum so that the packet will not be dropped at its next hop due to an invalid checksum. In IPv4, a recalculation of the layer 4 checksum is needed when layer 4 information is manipulated. The new checksum is based off of the translated layer 4 information. Following the checksum recalculation for layer 4, the NAT code manipulates the IP addresses and re-computes the IP header checksum. This logic flow works for ICMP for IPv4, however it does not work for ICMP for IPv6 (ICMPv6). The checksum for ICMPv6 is different from that for ICMP because it includes in its calculations not only the checksum of the layer 4 header information and the layer 4 data, but also the checksum of a pseudo-header. This pseudo-header is needed because there is no checksum in the IPv6 header to protect the header information. The pseudo-header consists of the source and destination IP address, the length of the layer 4 header and packet, the checksum field, and the next header field. Figure 12 shows the pseudo header format for IPv6 as taken from RFC 2460. [IP6 SPEC]

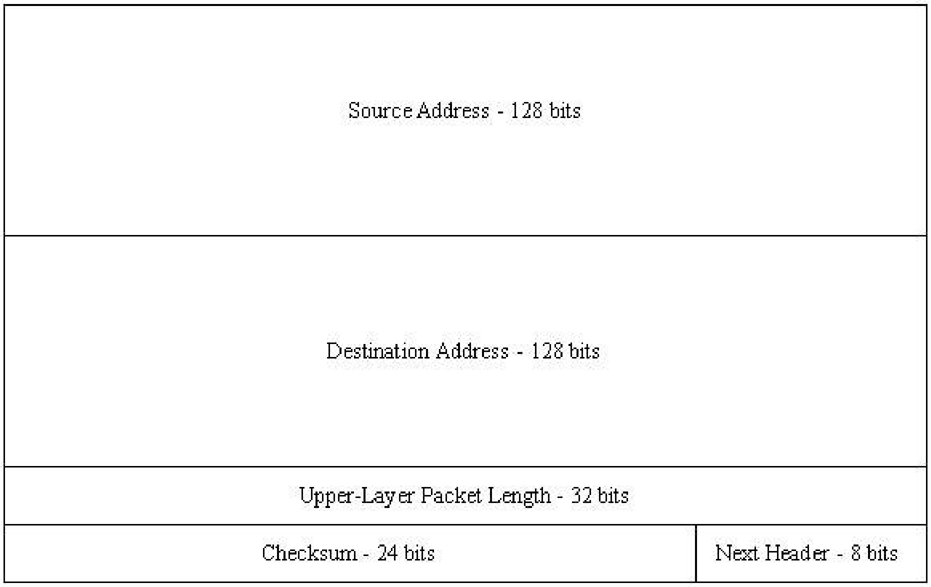
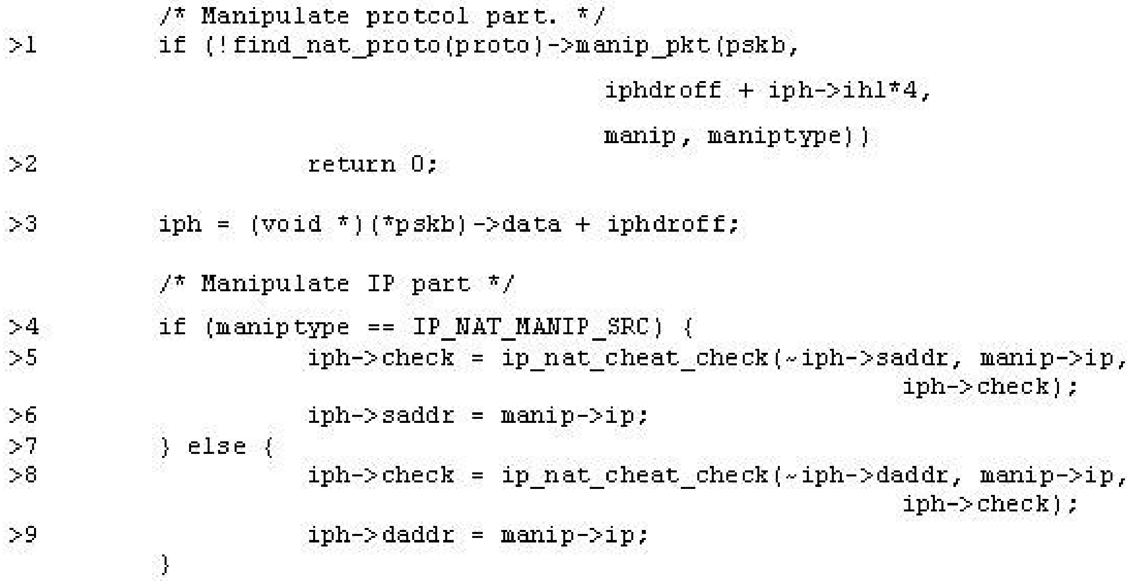

Source Address - 128 bits
Destination Address - 128 bits
Upper-Layer Packet Length - 32 bits
Checksum - 24 bits
Next Header - 8 bits

Figure 12. Layer 4 Pseudo-header for IPv6 [IP SPEC]

Calculation of the ICMPv6 checksum before IP address translation results in an incorrect checksum due to the IP addresses in the pseudo-header. The solution to this problem was to switch the order of the IP address translation and the layer 4 checksum calculation. Figure 13 shows a portion of the IPv4 manip_pkt() function, located in ip_nat_core.c, that recalculates the checksum of the IP header.



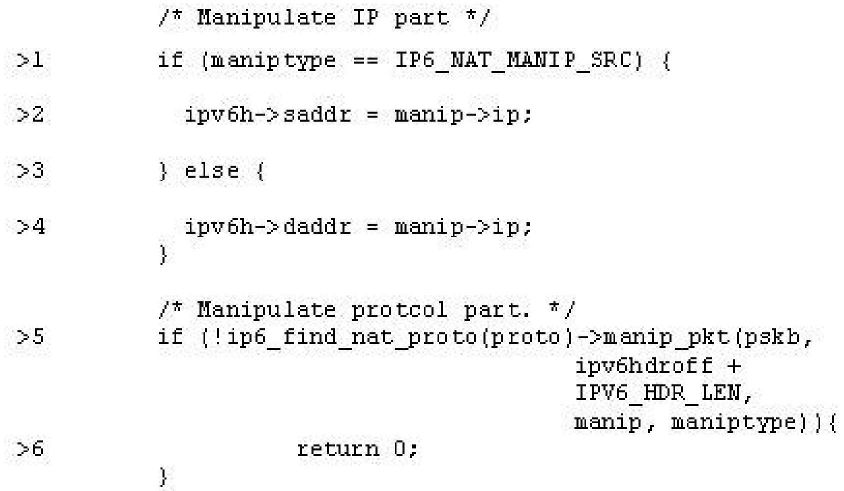
```
            /* Manipulate protcol part. */
>1          if (!find_nat_proto(proto)->manip_pkt(pskb,
                                          iphdroff + iph->ihl*4,
                                          manip, maniptype))
>2                  return 0;

>3          iph = (void *)(*pskb)->data + iphdroff;

            /* Manipulate IP part */
>4          if (maniptype == IP_NAT_MANIP_SRC) {
>5                  iph->check = ip_nat_cheat_check(~iph->saddr, manip->ip,
                                                        iph->check);
>6                  iph->saddr = manip->ip;
>7          } else {
>8                  iph->check = ip_nat_cheat_check(~iph->daddr, manip->ip,
                                                        iph->check);
>9                  iph->daddr = manip->ip;
            }
```

Figure 13. IPv4 Function manip_pkt()

In IPv4, the ICMP protocol information is manipulated (line 1) before the IP address is changed (lines 6 or 9 depending on the type of NAT) because the ICMP checksum did not depend on the IP addresses. For IPv6, the calculation of the pseudo-header in the ICMPv6 checksum requires a switch in logic so that the modified IP addresses can be included in the ICMP pseudo header checksum calculation. This change can be seen in Figure 14, which is the IPv6 version of the manip_pkt() function, located in ip6_nat_core.c



```
          /* Manipulate IP part */
>1        if (maniptype == IP6_NAT_MANIP_SRC) {

>2          ipv6h->saddr = manip->ip;

>3        } else {

>4          ipv6h->daddr = manip->ip;
          }

          /* Manipulate protcol part. */
>5        if (!ip6_find_nat_proto(proto)->manip_pkt(pskb,
                                        ipv6hdroff +
                                        IPV6_HDR_LEN,
                                        manip, maniptype)){
>6                return 0;
          }
```

Figure 14. IPv6 Function manip_pkt()

The “Manipulate IP part” (line 2 or 4 depending on type of NAT) and the “Manipulate protocol part” (line 5) are switched compared to the existing IPv4 NAT code. This change in ordering provided the pseudo-header calculations with the translated IP addresses necessary to calculate a valid checksum. Another noticeable difference between Figure 13 and Figure 14 is the absence of an IP header checksum calculation, which is present on lines 5 and 8 in Figure 13, but not needed by IPv6 (Figure 14).

3. Checksum Calculation Algorithm

In addition to changing the order of operations in the manip_pkt function, it was also necessary to alter the method in which checksums were calculated for the layer 4 headers. In IPv4, checksums of translated packets are calculated using an optimized algorithm implemented in ip_nat_cheat_check that uses bitwise manipulation to re-calculate the checksum based only on the changed ports and IP addresses. This function performs basic bit manipulations in assembly code and assumes that the input arguments are 32-bit integers. This assembly code function was not easily ported to support IPv6 data structures. A less optimized, but straightforward, method that uses the existing csum_partial() and csum_ipv6_magic() functions solved the problem. The csum_partial() function calculates the layer 4 checksum with the exception of bitwise manipulation. Omitting the final bitwise flip then allows the result to be folded into the pseudo-header checksum calculations done by csum_ipv6_magic(). This results in a valid layer 4 checksum which accounts for the pseudo-header, layer 4 header, and layer 4 payload.

D. DEBUGGING

After the initial one-to-one port was completed and successfully compiled into the kernel, the code did not function as expected. Debugging was necessary to determine why the code was not operating properly and extensive use of printk was the primary method for debugging the code. Printk is the kernel debugging mechanism that flushes debugging messages to a log file that a user can access. The initial solution to the improper code execution was to increase the number of debugging messages so that detailed call chains could be mapped out. These additional debugging statements allowed the code to be traced to within a function call of the problem.

One drawback of printk is that it requires that the kernel messages be flushed to the log file. This limitation resulted in a problem during testing, when an errant pointer caused the kernel to trap and lock up the system. To obtain real-time kernel debugging messages, and to allow the point of failure to be isolated, it was necessary to enable the serial console interface of the kernel. This interface allowed the display of real-time debugging messages on a remote console. This helped to isolate the trapping code.

While effective in displaying function call chains and variable values, the use of printk to display the actual packet data was cumbersome. To obtain this information it was necessary to use the tcpdump program to obtain a hexadecimal output of packets entering and leaving the TPE interfaces. This output allowed detailed scrutiny of packet contents to ensure that information into and out of the TPE was correct. Tcpdump places the network interface card (NIC) into the promiscuous mode to capture all packets entering or leaving the interface, and outputs the captured information. This simplicity and its inclusion as a standard tool in almost all Linux distributions made tcpdump a good choice for packet capturing on the TPE.

During the later stages of debugging and development the TPE would properly translate IP addresses, but the packet would be dropped at the server. This problem was approached by placing the Ethereal program on both the client and server machines. Ethereal produced a much more detailed and user-friendly output. This output helped solve the question as to why packets would reach the server properly translated, yet still be dropped. Ethereal output showed that the packet was properly translated but that its checksum was incorrect. Ethereal also calculated what the checksum should have been given the header information. The reason packets were being dropped at the server was an improper checksum calculated by the NAT code, because the wrong length was passed to the checksum functions. Once this problem was fixed, packets were successfully transmitted and the NAT code functioned properly.

E. TESTING

For this thesis, different network applications were used to verify that the SNAT implementation works correctly. Three types of protocols were tested: ICMPv6, UDP, and TCP. ICMPv6 was tested using the *ping6* mechanism, UDP was tested using the *traceroute6*, and TCP was tested using *rlogin* and by downloading a webpage through the TPE. During each testing phase various errors were encountered and ultimately fixed.

Ping6 tested ICMPv6 NAT by sending an Echo Request packet from the Client to the Server. This Echo Request packet then resulted in transmission of an Echo Reply packet back to the client. During this testing Ethereal showed that the ICMPv6 Echo Request packet was properly translated through the TPE, but was dropped by the server. This problem was a result of the improper calculation of the checksum and the calculation of the checksum before the IP address was translated. Once this logic and ordering was fixed, *ping6* completed successfully. Ethereal showed proper translation of both the ICMPv6 Echo Request and Echo Reply packets to and from the client and server machines. These Ethereal and *tcpdump* outputs can be found in Appendix D.

The next test was to run traceroute6 to test UDP packet translation. Traceroute6 determines the hop-by-hop route from source to destination. A UDP packet is sent out with a hop limit of one and each time *traceroute6* receives an ICMPv6 Timeout message, it records the IP address and adds it the route. When the UDP packet reaches the specified target destination, an ICMPv6 Destination Unreachable message is returned and the route is displayed.

During NAT testing, Ethereal output from the server showed that while UDP packets were properly translated, the replying ICMPv6 Destination Unreachable packet was dropped at the TPE. After finding a misplaced bracket in the ported code, the ICMPv6 Destination Unreachable packet was properly forwarded from the server, through the TPE, to the client.

At this point, the Ethereal output from the client showed that the ICMPv6 Destination Unreachable packet was being dropped due to an improper checksum. Further evaluation of the call chain, and respective code, revealed that ICMPv6 error messages followed a different logic flow. This logic flow reached a checksum calculation, in the function icmpv6_reply_translation() located in ip6_nat_core.c, that had not been changed to use csum_partial() and csum_ipv6_magic(). After the checksum calculation logic was fixed, the client properly received the ICMPv6 Destination Unreachable packet from the Server, and traceroute6 operated successfully.

The NAT mechanism for TCP uses the same logical flow as UDP packets except that the TCP flow also accounts for ports which is beyond the scope of this thesis. After changing the TCP checksum calculation to use csum_partial() and csum_ipv6_magic(), TCP packets properly traversed the NAT mechanism in the TPE. To test the translation of TCP packets from the client to the server and back again, rlogin was used in addition to downloading a webpage. Using *rlogin*, an authorized user was able to log into the server from the client, list a directory, use "cat" to list the contents of a text file, and then log off, all through the TPE running NAT. Downloading the webpage involved setting up an Apache web server on the Server machine and then downloading the webpage through the TPE running NAT. The results of both tests were verified by Ethereal outputs (see Appendix D) that showed the proper translations at both the server and client ends of the IPv6 NAT test bed.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSION & FUTURE WORK

This chapter gives an analysis of the NAT mechanism integrated within the Linux kernel for use with IPv6. Recommendations for future work on the IPv6 NAT mechanism and suggestions for pursuit of future work on the current NAT implementation for IPv6 within the Linux kernel are also presented.

A. ANALYSIS OF THE INTEGRATED NAT

Upon completion of debugging, the NAT mechanism was tested within the framework of the IPv6 NAT tested for use with the MYSEA architecture. As explained in Chapter IV, the NAT mechanism was placed on the TPE and all traffic from the client or server must pass through the TPE. Testing of three protocols, TCP, UDP and ICMP, was conducted and the results are described in Chapter 4 and can be found in Appendix D. The SNAT functionality was demonstrated to function properly with use of common networking applications such as *ping6*, *traceroute6* and *rlogin*. As a culminating experiment, an Apache web server was placed on the testbed server and hosted multiple web pages. The IPv6 NAT mechanism was then activated on the TPE with rules set to mask the identity of the client. The client then accessed the web pages from the server, through the TPE, with a connection that was successfully masked through the NAT mechanism. Ethereal was used to verify the successful translation of packets.

B. FUTURE ALTERNATE IMPLEMENTATION DESIGN

The NAT mechanism developed in this thesis is based on the current dual-stack architecture in current Linux kernel releases. In this architecture, there are separate call chains for IPv4 and IPv6. For future NAT developments, there are two possibilities for implementation redesign: rewrite the current dual-stack *netfilter* architecture into a single stack, or decouple the NAT functionality from *netfilter*.

Currently, there are two separate *netfilter* stacks for IPv4 and IPv6. This functionality traces to the initialization of *netfilter* where it is called by different receive functions for IPv4 packets and IPv6 packets. A future design could combine the functionality present in both stacks into one, cohesive stack. This would reduce the amount of functionality duplication. This is beneficial for assurance purposes since there would be less code to verify. Also, the code would be more efficient and require less memory than the current implementation. This project would however involve an immense amount of work and a great deal of previous knowledge regarding both *netfilter* and Linux kernel programming.

Alternately, the NAT mechanism could be removed from *netfilter* such that it is a separate entity. Presently, the NAT mechanism is highly dependant on the *netfilter* architecture.

Another redesign alternative would be to develop a kernelized NAT mechanism that operates in a completely isolated manner. Here, the mechanism would most likely intercept the packet before *netfilter* manipulates it via the receive functions. This design would be beneficial since all the NAT functionality would be modular and thus better suited for a high assurance design.

C. OTHER FUTURE WORK

There are many areas in which the current implementation could be improved without restructuring the architecture or design. These include but are not limited to work on: extension headers, multiple protocol support, greater user-space functionality, port translation, multiple types of NAT mechanisms, and address or port ranges.

One recommendation is to fix the checksum calculations to handle extension headers when calculating the length of layer 4. In the current implementation, this is calculated by subtracting the IPv6 header length from the length field in the skb. This has the potential to yield an improper value, specifically if extension headers are present. In the current protocol, the *next header* field of the IPv6 header does not necessarily point directly to layer 4, primarily in the case of extension headers resulting from IPSEC or ESP. [IP SPEC] A suggested method for properly calculating the layer 4 payload length would be parse individual fields from the skb until the beginning of the layer 4 header is reached and then calculate the length.

Another recommendation for future development would be to enable support for other layer 4 protocols. Currently, the implementation supports TCP, ICMP and UDP. Though these protocols enable a great deal of functionality, there are other protocols, such as FTP, TFTP and IRC, that are functional in IPv4 that are not yet developed for IPv6. For example, IRC will be required to support the new Naval Research Laboratory multilevel chat program in the MYSEA multilevel testbed.

At this time, only the SNAT target has been ported to the IPv6 user-space *ip6tables*. For the purposes of this thesis within the IPv6 NAT testbed for the MYSEA architecture, it was the only target needed for testing purposes. There are other targets, such as DNAT and MASQUERADE, which are present in IPv4 that have yet to ported to IPv6. Porting these targets would allow greater flexibility for the *nat* table within *ip6tables*.

Future work could involve advancing the current NAT implementation to perform the additional NAT functions as dictated by RFC 2663. [IPNATTC] This NAT implementation only handles the functionality necessary for basic NAT. [IPNATTC] This work could involve adding port translation, destination NAT and static NAT support to the current NAT implementation.

Finally, this NAT implementation does not support the assignment of ranges of either ports or addresses for the address translation mechanism to use. Future work in this area could include not only developing the user space and kernel space to accept ranges, but also the development of a robust algorithm for use in assigning either addresses or ports.

D. SUMMARY

A working implementation of NAT for IPv6 within the Linux kernel has been produced. It was created on a modified version of the Linux 2.6.5 Kernel that supports connection tracking. The NAT development created here has been tested for support of the protocols of TCP, UDP and ICMP for IPv6.

LIST OF REFERENCES

[NRD] Arkko, J. "Securing IPv6 Neighbor and Router Discovery." Proceedings of the ACM Wireless Security Workshop. September 2002.

[SPIRAL] Boehm, B. "A Spiral Model of Software Development and Enhancement." IEEE Computer. May 1988.

[CC] Common Criteria for Information Technology Security Evaluation, Part 3: Security Assurance Requirements. Version 2.2, Revision 256. CCIMB-2004-01-003. January 2004.

[CC SECEVAL] Cox, P. "Security Evaluation: The Common Criteria certifications." [http://www.itsecurity.com/papers/border.htm]. 2000.

[IP6 ADDR] Deering S., and Hinden R. "Internet Protocol Version 6 (IPv6) Addressing Architecture," RFC 3513. April 2003.

[IP6 SPEC] Deering S. and Hinden R. "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460. December 1998.

[MTU] Deering, S. and Mogul, J. "Path MTU Discovery," RFC 1191. November 1990.

[MEMO] Department of Defense memorandum, SUBJECT: Internet Protocol version 6 (IPv6). June 9, 2003.

[NGIP] Drakos, N. "Next Generation IP" [http://www.cs.arizona.edu/llp/book/node53.html]. December 1995.

[NF OVER] Eastep, T. "Netfilter Overview." [http://www.shorewall.net/NetfilterOverview.html]. March 2004.

[ACM IP6] Gulati, S. "The Internet Protocol – Part II : The Present and the Future." [http://www.acm.org/crossroads/columns/connector/august2000.html]. January 2001.

[CC WWC] Hayes, K. "Common Criteria – A World Wide Choice." [http://www.itsecurity.com/papers/88.htm]. 1998.

[IPng] Hinden, R. "IP Next Generation Overview" Communications of the ACM. June 1996.

[IP6 URL] Hinden, R. and others "Format for Literal IPv6 Addresses in URL's," RFC 2732. December 1999.

[NONAT] IPv6 Forum. "NAT: Just Say No." [http://www.circleid.com/article/355_0_1_0_C/]. October 2003.

[MYSEA] Irvine, C., and others "Overview of a High Assurance Architecture for Distributed Multilevel Security." Proceedings of the 2004 IEEE Workshop on Information Assurance and Security. June 2004.

[DoD SOIP] Kent, S. "U.S. Department of Defense Security Options for the Internet Protocol," RFC 1108. November 1991.

[MYSEA COMP] O'Neal, M. *A Design Comparison Between IPv4 and IPv6 in the Context of MYSEA, and Implementation of an IPv6 MYSEA Prototype.* Master's Thesis. Naval Postgraduate School. Monterey, California. June 2003.

[ICMP SPEC] Postel, J. "Internet Control Message Protocol," RFC 792. September 1981.

[IP] Postel, J. “Internet Protocol,” RFC 791. September 1981.

[AAPI] Rekhter, Y. and others “Address Allocation for Private Internets,” RFC 1918. February 1996.

[NFHH] Russell, R. and Welte, H. “Linux netfilter Hacking HOWTO.” [http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html]. July 2002.

[IPNATTC] Srisuresh, P. and Egevang, K. “IP Network Address Translator (NAT) Terminology and Considerations,” RFC 2663. August 1999.

[TNAT] Srisuresh, P. and Egevang, K. “Traditional IP Network Address Translator (Traditional NAT),” RFC 3022. January 2001.

[MOSIX] Subramanian, R. *A Technique for Improving the Scheduling of Network Communicating Processes in MOSIX*. Master’s Thesis. Kansas State University. Manhattan, Kansas. December 2002.

[KERB] Thomas, M. “Requirements for Kerberized Internet Negotiation of Keys,” RFC 3129. June 2001.

[IPROUTE] Waldvogel, M., and others “Scalable High Speed IP Routing Lookups.” ACM SIGCOMM 1997 Conference. 1997

[NGI] Weiser, M. “What Ever Happened to the Next Generation Internet?” Communications of the ACM. September 2001.

[LNF] Wronkowski, M. “Linux Netfilter.” [http://www.csh.rit.edu/~mattw/proj/nf/].

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. CHANGE CONTROL PROCEDURES

VERSION CONTROL AND BACKUP PLAN

A standard naming scheme was developed that allowed versions to be tracked as well as the restoration of previous versions, should that become necessary. The naming scheme is as follows: NAME-MM-DD-YYYY-V, where NAME is the name of the document, MM is the month, DD is the day, YYYY is the year, and V is the version for that particular day using the alphabet (ie, ver A, ver B, etc.). This allowed versions to be found easily and changes to be tracked throughout the development process.

The backup plan was fairly simple as well. Upon creation of a new version of any document, the first step was to save the document locally on either the network drive or the home computer. The next immediate step was to email the document to the thesis partner and the originator of the document, effectively storing a copy on the mail server. Additionally, an archive of all thesis related documents was compiled on writeable CD/DVD media, and on home computers as needed. This provided sufficient redundancy, and given the version control scheme, it permitted fairly easy recovery from any loss of data. In the event of data loss, the procedure would have been to copy the archive over to the affected machine.

In addition to the backup plan, multiple systems were maintained on which the most up to date files and pieces of code could be found. The client machine in the lab was configured to dual boot into either Windows XP or Red Hat Linux. Both of these partitions served as repositories for thesis documents. The source code for the project was stored on both the TPE and on writeable CD. In addition, versions of the thesis were stored on personal workstations, USB removable storage and writeable CD.

CONFIGURATION ITEMS AND DESCRIPTION

This project included all the listed files ported or altered in order to obtain working NAT functionality. A distinction was made between ported and altered files. Altered files were existing files that required modification in order to support NAT. Ported files were files that did not exist in the working 2.6.5 kernel with IPv6 connection tracking, and were necessary to obtain NAT functionality. This code was comprised mostly of NAT files ported to IPv6. The following is a list of all the altered or created files within the *netfilter* suite:

- /include/linux/netfilter_ipv6/ip6_conntrack.h
- /net/ipv6/netfilter/ip6_conntrack_core.c
- /include/linux/netfilter_ipv6/ip6_nat.h
- /net/ipv6/netfilter/ip6_nat_core.c
- /include/linux/netfilter_ipv6/ip6_nat_core.h
- /net/ipv6/netfilter/ip6_nat_helper.c
- /include/linux/netfilter_ipv6/ip6_nat_helper.h
- /net/ipv6/netfilter/ip6_nat_proto_icmp.c
- /net/ipv6/netfilter/ip6_nat_proto_tcp.c
- /net/ipv6/netfilter/ip6_nat_proto_udp.c
- /net/ipv6/netfilter/ip6_nat_proto_unknown.c
- /include/linux/netfilter_ipv6/ip6_nat_protocol.h
- /net/ipv6/netfilter/ip6_nat_rule.c
- /include/linux/netfilter_ipv6/ip6_nat_rule.h
- /net/ipv6/netfilter/ip6_nat_standalone.c

- /include/linux/netfilter_ipv6/ip6t_iprange.h
- /net/ipv6/netfilter/ip6t_NETMAP.c
- /net/ipv6/netfilter/ip6t_SAME.c
- /include/linux/netfilter_ipv6/ip6t_SAME.h
- /net/core/netfilter.c
- /home/iptables-1.2.9rc1/extensions/libip6t_SNAT.c

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. SPECIFICATION DOCUMENT

INTRODUCTION

The Network Address Translation (NAT) for IPv6 will be developed using a modified Linux 2.6.5 kernel that supports connection tracking. Intended users for this application are any user desiring IPv4 NAT functionality for IPv6. Specifically, these are users desiring to translate addresses from a private network to authorized public network addresses.

The main purpose of this application is to provide NAT functionality for IPv6. NAT only deals with altering the IP header fields and checksums in the IPv6 datagram packets. Additionally, an interface for interaction with the NAT to allow static binding of IP addresses and assignment of dynamic IP address ranges is desired. User and system interactions include calls to and from the kernel module, calls to the *ip6tables*, calls to the *netfilter* module, which handles packet processing in general, calls to connection tracking, and reference to information in the NAT table.

Users will interact with the protocol from a command line interface by altering the *nat* table entries to reflect desired translations. Additional software requirements include the use of Application Level Gateways (ALGs) to help any software that alters IP information interact with the NAT device. These gateways would be designed and implemented by the producers of the given software.

OPERATING ENVIRONMENT

The expected operating environment of this NAT implementation will be on a networked computer running a modified version of Red Hat 9.0 design to support connection tracking interacting with an IPv6 network. Initially the environment will be a closed, with the NAT mechanism performing a one-to-one translation; however, ultimately the implementation will be usable by any person running the modified Linux 2.6.5 kernel wishing to run NAT on IPv6. Off-the-shelf tools will be the modified 2.6.5 Linux kernel designed to support connection tracking and a personal computer capable of being networked and of running the operating system and capable of networking with an IPv6 network.

The computer running the NAT protocol should be of sufficient speed to perform the address translation without any noticeable delay or hindrance to network communications. The physical environment of the protocol will be constrained by the physical hardware needed to implement the NAT protocol. Namely, the requirements the physical computer and networking devices have will also be those of the application. Should this application be deployed in an untrusted environment, special care must be take to safeguard the NAT device so that it is not turned off or manipulated, allowing external networks to communicate directly with the internal networks using their true IP address.

Users are expected to understand the basics of both the Linux operating system and IP networking. The user must know what NAT does and how it performs its job. Expected usage pattern of the NAT protocol will be that of the network.

INTERFACES

Operation of NAT should be fairly transparent to the user, therefore only a simple interface to allow static binding of IP addresses will be provided. It will allow the NAT protocol to be turned on or off, and will allow the user to program desired translations into the *nat* table. The existing *ip6tables* will be used as the interface.

Access to the NAT interface will be limited to users with root privilege.

The interface will be command line, since it was previously implemented in command line in IPv4 and this would appear to be the simplest and most efficient method of interaction.

The interface will consist of commands that allow the user to perform the tasks of configuring translations, static bindings, and turning NAT off and on. The interface will only manipulate the *nat* table and allow the user to start and stop the NAT. Since there will be only one interface, there will not be any inter-interface dependencies. NAT information will be transferred to and from the interface as soon as it is updated so that the interface gives the user accurate information of what the *nat* table contains. The NAT interface will still be functional when there is no networking connection because the user can still set up a *nat* table and NAT rules regardless of whether or not there is connectivity.

SYSTEM OPERATION

The runtime protocol will operate in a passive mode. Its presence should be transparent to the user. NAT operation will begin after the user invokes the protocol through the interface, which will activate the NAT functionality within the *netfilter* hooks. An overview of the protocol’s operation is characterized diagrammatically by the following flow chart. (See Figure 15) Note that in the diagram the firewall system merely refers to the computer that receives the packet.

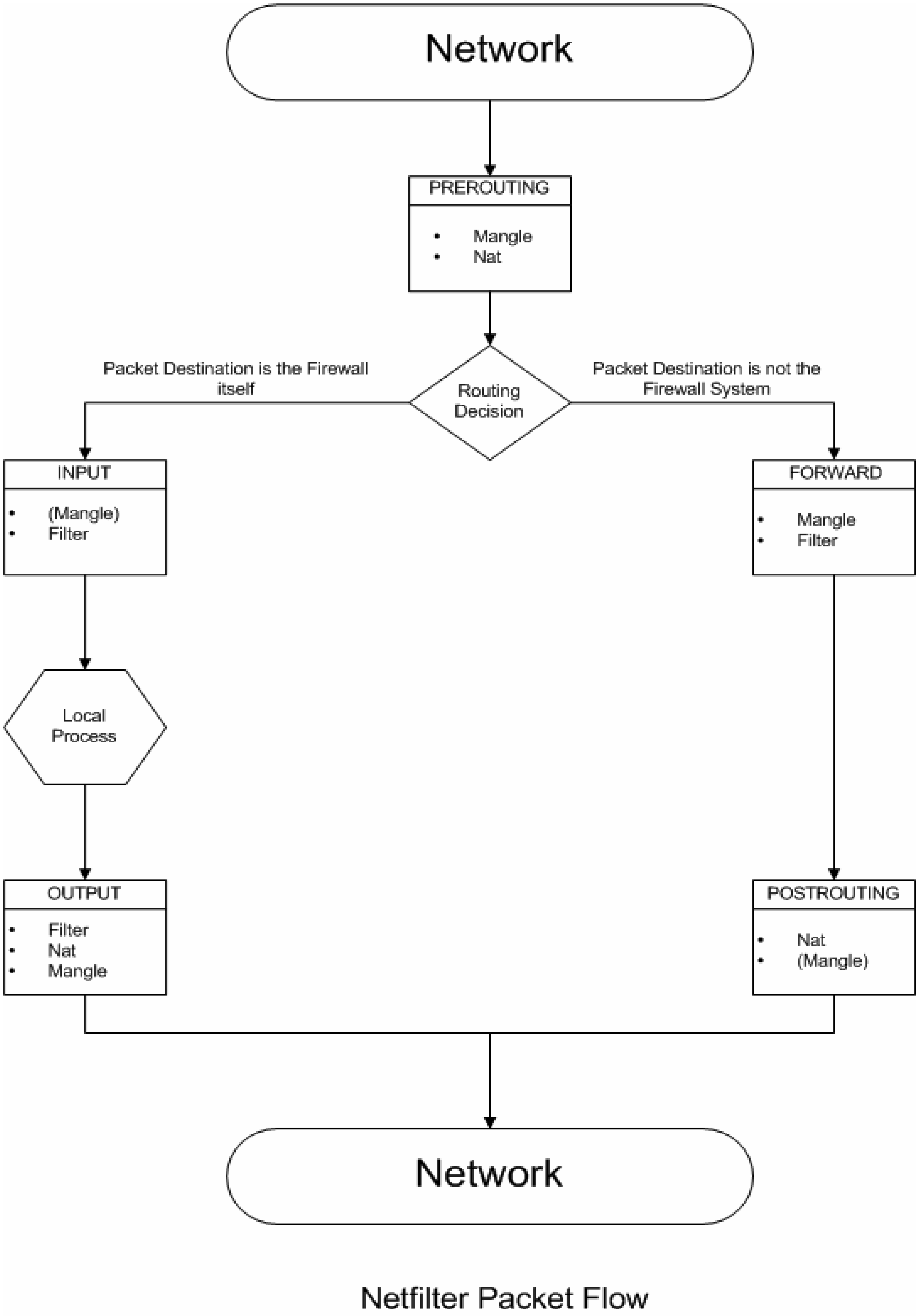


Figure 15. Netfilter Packet Flow [NF OVER]

The user interface alters the *nat* table rules that are traversed when it is called by one of the *netfilter* hooks.

The application of this development is restricted to a networking environment. There is no necessity for NAT in a stand-alone environment. Major components of the application include: the interface, the *nat* table, any modified kernel source code, *netfilter* and its hooks, and the user interface to *netfilter* and *ip6tables*. The *nat* table will be stored in a non-volatile location to eliminate the necessity of re-entering the translation mappings every time the machine is rebooted.

DATA TYPES & STORAGE

The NAT protocol for IPv6 will use the source and destination addresses, the IP header checksum, and the *nat* table, which stores the mappings. All other information within the IPv6 header, while related to the task of NAT, is not specific to what the NAT protocol will do. The source and destination address both tell the IP packet where to go, and in the case of this implementation, they will be replaced by desired mappings to hide the true IP address from the sender or receiver, depending on the type of NAT employed, preventing them from gaining privileged knowledge of the network topography.

The source and destination addresses will be stored by the connection tracking module and read by the *netfilter* hooks to determine if any rules exist in the *nat* tables for the specific IP conversation. The IP address will then be translated by *ip6tables*. The layer 4 header checksum is a quality of service mechanism that helps assure that the original packet has remained unchanged. This checksum will be invalid if either the source or destination IP addresses are modified, unless the checksum is recalculated to reflect the changed IP addresses. The NAT mechanism will recalculate this checksum in a manner dependant on the protocol of the packet traversing the NAT mechanism. The mappings contained in the *nat* table will be compared by *ip6tables* to the session information saved in connection tracking to see if rules need to be applied.

PERFORMANCE

A majority of the performance requirements were listed in the above sections. Basically, the *ip6tables* will interact with the NAT code in a manner that does not prove to be a hindrance to network operations. The exact threshold for this is not static, but rather it varies from user to user, since a network administrator with a gigabit Ethernet LAN may have higher performance requirements than a home user with a small LAN connecting to the Internet via a 56K modem.

The maximum number of concurrent users will be the number of users within the administrator group, as the NAT rules are only able to altered by a user with administrator privilege. In general there will be no necessity for multiple users to alter the *nat* tables, unless there is some sort of cooperative environment agreed upon by multiple users of the LAN. Also, any user that requests to alter the *nat* table must have administrator privilege.

The maximum number of concurrent connections will be limited by the maximum number of ports available multiplied by the number of public IP addresses the NAT device maintains for external translation. The expected usage pattern will be constant. Once configured and operational, the only further alterations to the *nat* table should be when external IP address bindings are reconfigured.

The tolerance for error will be fairly low, as any error in translation will result in undesired network operation and the probable loss of connectivity to the Internet. Workload expectations for the protocol will depend on the amount of network traffic passing through the computer. Critical resources for this program are Internet connectivity and adequate processor speed.

PARALLELISM

This NAT development does not require any parallelism, as it is an in-line function. When a packet enters the NIC interface *netfilter* hooks are called. These hooks traverse a list of processes that have requested access to the packet in a priority queue. One of these processes is always *ip6tables*, and within *ip6tables* is the NAT code. While the NAT process is running and manipulating the packet, there will not be any other processes manipulating the packet simultaneously.

CONCURRENT ENGINEERING

There will not be any concurrent engineering with respect to development, testing, and deployment of this program. As was stated with parallelism, there is no necessity for parallel access, engineering, or development.

SECURITY

The process will have all of the security characteristics of *ip6tables*, the user interface to *netfilter*. Currently, *ip6tables* cannot be edited unless the user has administrator privileges. Therefore, the NAT process will not be accessible to any user other than root or those users with root permissions. Allowing any users to edit any of the tables for *ip6tables* would leave the system open to any number of security violations as a malicious user could set mappings and intercept traffic, as well as, masquerade as any user on the LAN.

IMPLEMENTATION PLAN

Development of NAT for IPv6 will occur on a modified 2.6.5 Linux kernel designed to support connection tracking. The necessary IPv4 NAT code will be ported and modified into the IPv6 environment. Initially, the focus was to enable connection tracking for IPv6 before the development of the NAT functionality. However, a modified 2.6.5 Linux kernel was released that enabled this functionality. From this point, the *nat* table to the *ip6tables* code and its respective functionality will be introduced. After this foundation is laid, the desired NAT functionality will be implemented in a method similar to IPv4. Finally, open-source testing suites will be used to test module compatibility, and the MYSEA IPv6 NAT testbed will be used as method for testing the functionality.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. SOURCE CODE

This appendix contains all the source code for files within the Linux netfilter suite for IPv6 that were either altered or created in order to support NAT in IPv6. The altered files contain the inserted code. The created code contains a header that declares it as such.

/INCLUDE/LINUX/NETFILTER_IPV6/IP6_CONNTRACK.H

```
/*
 * Copyright (C)2003 USAGI/WIDE Project
 *
 * Authors:
 *    Yasuyuki Kozakai  <yasuyuki.kozakai@toshiba.co.jp>
 *
 * Based on: include/linux/netfilter_ipv4/ip_conntrack.h
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 *
 * 24-May-2004   : Ported application helper data union for later use -
TB MP
 *               : Ported NAT helper connection tracking data union for
later use - TB MP
 */


/* per expectation: application helper private data */
union ip6_conntrack_expect_help {
      /* insert conntrack helper private data (expect) here */
      struct ip6_ct_ftp_expect exp_ftp_info;


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * *
 * TB MP - This is where nat helper private data goes.  Not ported by
USAGI.  This was
 * ported, however it was not used because the thesis only deals with
basic NAT.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * */

#ifdef CONFIG_IP6_NF_NAT_NEEDED
      union {
            /* insert nat helper private data (expect) here */
      } nat;
#endif

  /*TB MP - END NAT CODE*/

};


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - Nat helper information for connection tracking goes here.
Not ported by USAGI.
 * This was ported, however it was not used because the thesis only
deals with basic NAT.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */

#ifdef CONFIG_IP6_NF_NAT_NEEDED
#include <linux/netfilter_ipv6/ip6_nat.h>

/* per conntrack: nat application helper private data */
union ip6_conntrack_nat_help {
      /* insert nat helper private data here */

};

#endif

/*TB MP - END NAT CODE*/
```

/NET/IPV6/NETFILTER/IP6_CONNTRACK_CORE.C

```
/*
 * IPv6 Connection Tracking
 * Linux INET6 implementation
 *
 * Copyright (C)2003 USAGI/WIDE Project
 *
 * Authors:
 *    Yasuyuki Kozakai  <yasuyuki.kozakai@toshiba.co.jp>
 *
 * Based on: net/ipv4/netfilter/ip_conntrack_core.c
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 *
 * 24-May-2004: Ported NAT code for ICMP tracking - TB MP
 *             : Ported NAT code to reverse connection direction - TB MP
 *              : Ported NAT function that expects a connection change -
TB MP
 */


struct ip6_conntrack *
icmp6_error_track(struct sk_buff *skb,
              unsigned int icmp6off,
              enum ip6_conntrack_info *ctinfo,
              unsigned int hooknum)
{

      struct ip6_conntrack_tuple intuple, origtuple;
      struct ip6_conntrack_tuple_hash *h;
      struct ipv6hdr *ip6h;
      struct icmp6hdr hdr;
      struct ipv6hdr inip6h;
      unsigned int inip6off;
      struct ip6_conntrack_protocol *inproto;
      u_int8_t inprotonum;
      unsigned int inprotoff;
      IP6_NF_ASSERT(skb->nfct == NULL);

      ip6h = skb->nh.ipv6h;
      if (skb_copy_bits(skb, icmp6off, &hdr, sizeof(hdr)) != 0) {
            DEBUGP("icmp_error_track: Can't copy ICMPv6 hdr.\n");
            return NULL;
      }

      if (hdr.icmp6_type >= 128){
            return NULL;
      }
      /*
       * Should I ignore invalid ICMPv6 error here ?
       * ex) ICMPv6 error in ICMPv6 error, Fragmented packet, and so
on.
       * - kozakai
```

```
         */

        /* Why not check checksum in IPv4 conntrack ? - kozakai */
        /* Ignore it if the checksum's bogus. */


        if   (csum_ipv6_magic(&ip6h->saddr,   &ip6h->daddr,   skb->len   -
icmp6off,
                        IPPROTO_ICMPV6,
                        skb_checksum(skb, icmp6off,
                                skb->len - icmp6off, 0))) {
                DEBUGP("ICMPv6 checksum failed\n");
                return NULL;
        }

        inip6off = icmp6off + sizeof(hdr);


        if (skb_copy_bits(skb, inip6off, &inip6h, sizeof(inip6h)) != 0) {
                DEBUGP("Can't copy inner IPv6 hdr.\n");
                return NULL;
        }

        inprotonum = inip6h.nexthdr;
        inprotoff = ip6_ct_skip_exthdr(skb, inip6off + sizeof(inip6h),
                                &inprotonum,
                                skb->len - inip6off - sizeof(inip6h));

        if (inprotoff < 0 || inprotoff > skb->len
            || inprotonum == NEXTHDR_FRAGMENT) {
                DEBUGP("icmp6_error:  Can't  find  protocol  header  in  ICMPv6
payload.\n");
                return NULL;
        }

        inproto = ip6_ct_find_proto(inprotonum);
        /* Are they talking about one of our connections? */
        if (!ip6_get_tuple(&inip6h, skb, inprotoff, inprotonum,
                       &origtuple, inproto)) {
                DEBUGP("icmp6_error: ! get_tuple p=%u\n", inprotonum);
                return NULL;
        }

        /* Ordinarily, we'd expect the inverted tupleproto, but it's
           been preserved inside the ICMP. */

        if (!invert_tuple(&intuple, &origtuple, inproto)) {
                DEBUGP("icmp6_error_track: Can't invert tuple\n");
                return NULL;
        }

        *ctinfo = IP6_CT_RELATED;
        h = ip6_conntrack_find_get(&intuple, NULL);
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - Code necessary for NAT that was not originally ported over
to IPv6 with the USAGI
 * connection tracking port.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */
      if (!h) {

            /* Locally generated ICMPs will match inverted if they
               haven't been SNAT'ed yet */
            /* FIXME: NAT code has to handle half-done double NAT --RR
*/
        if (hooknum == NF_IP6_LOCAL_OUT){
          h = ip6_conntrack_find_get(&origtuple, NULL);

          /*TB MP - END NAT CODE*/

        }
        if (!h) {
    DEBUGP("icmp6_error_track: no match\n");
          return NULL;
        }


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - Code necessary for NAT that was not originally ported over
to IPv6 with the USAGI
 * connection tracking port.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */

        /*Reverse direction from that found */
        if (DIRECTION(h) != IP6_CT_DIR_REPLY){
          *ctinfo += IP6_CT_IS_REPLY;
        }
        /*TB MP - END NAT CODE*/


       } else {
          if (DIRECTION(h) == IP6_CT_DIR_REPLY){
            *ctinfo += IP6_CT_IS_REPLY;
          }
       }

      /* Update skb to refer to this connection */
      skb->nfct = &h->ctrack->infos[*ctinfo];
      return h->ctrack;
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - Code necessary for NAT that was not originally ported over
to IPv6 with the USAGI
 * connection tracking port.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * */
int ip6_conntrack_change_expect(struct ip6_conntrack_expect *expect,
                         struct ip6_conntrack_tuple *newtuple)
{
      int ret;
      MUST_BE_READ_LOCKED(&ip6_conntrack_lock);
      WRITE_LOCK(&ip6_conntrack_expect_tuple_lock);
      DEBUGP("change_expect:\n");
      DEBUGP("exp tuple: "); DUMP_TUPLE(&expect->tuple);
      DEBUGP("exp mask:  "); DUMP_TUPLE(&expect->mask);
      DEBUGP("newtuple:  "); DUMP_TUPLE(newtuple);
      if (expect->ct_tuple.dst.protonum == 0) {
            /* Never seen before */
            DEBUGP("change expect: never seen before\n");
            if (!ip6_ct_tuple_equal(&expect->tuple, newtuple)
                && LIST_FIND(&ip6_conntrack_expect_list, expect_clash,
               struct ip6_conntrack_expect *, newtuple, &expect->mask))
            {

                  /* Force NAT to find an unused tuple */
                  ret = -1;
            } else {
                  memcpy(&expect->ct_tuple,              &expect->tuple,
                  sizeof(expect->tuple));
                  memcpy(&expect->tuple,    newtuple,    sizeof(expect-
                  >tuple));
                  ret = 0;
            }
      } else {
            /* Resent packet */
            DEBUGP("change expect: resent packet\n");
            if (ip6_ct_tuple_equal(&expect->tuple, newtuple)) {
                  ret = 0;
            } else {
                  /* Force NAT to choose again the same port */
                  ret = -1;
            }
      }
      WRITE_UNLOCK(&ip6_conntrack_expect_tuple_lock);
      return ret;
}

/* TB MP - END NAT CODE*/
```

/INCLUDE/LINUX/NETFILTER_IPV6/IP6_NAT.H

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: include/linux/ip_nat.h
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

#ifndef _IP6_NAT_H
#define _IP6_NAT_H
#include <linux/netfilter_ipv6.h>
#include <linux/netfilter_ipv6/ip6_conntrack_tuple.h>

#define IP6_NAT_MAPPING_TYPE_MAX_NAMELEN 16

enum ip6_nat_manip_type
{
      IP6_NAT_MANIP_SRC,
      IP6_NAT_MANIP_DST
};

#ifndef CONFIG_IP6_NF_NAT_LOCAL
/* SRC manip occurs only on POST_ROUTING */
#define HOOK2MANIP(hooknum) ((hooknum) != NF_IP6_POST_ROUTING)
#else
/* SRC manip occurs POST_ROUTING or LOCAL_IN */
#define  HOOK2MANIP(hooknum)  ((hooknum)  !=  NF_IP6_POST_ROUTING  &&
(hooknum) != NF_IP6_LOCAL_IN)
#endif

#define IP6_NAT_RANGE_MAP_IPS 1
#define IP6_NAT_RANGE_PROTO_SPECIFIED 2
/* Used internally by get_unique_tuple(). */
#define IP6_NAT_RANGE_FULL 4
```

```
/* NAT sequence number modifications */
struct ip6_nat_seq {
      /* position of the last TCP sequence number
       * modification (if any) */
      u_int32_t correction_pos;
      /* sequence number offset before and after last modification */
      int32_t offset_before, offset_after;
};

/* Single range specification. */
struct ip6_nat_range
{
      /* Set to OR of flags above. */
      unsigned int flags;

      /* Inclusive: network order. */
      struct in6_addr min_ip, max_ip;

      /* Inclusive: network order */
      union ip6_conntrack_manip_proto min, max;
};

/* A range consists of an array of 1 or more ip6_nat_range */
struct ip6_nat_multi_range
{
      unsigned int rangesize;

      /* hangs off end. */
      struct ip6_nat_range range[1];
};

/* Worst case: local-out manip + 1 post-routing, and reverse dirn. */
#define IP6_NAT_MAX_MANIPS (2*3)

struct ip6_nat_info_manip
{
      /* The direction. */
      u_int8_t direction;

      /* Which hook the manipulation happens on. */
      u_int8_t hooknum;

      /* The manipulation type. */
      u_int8_t maniptype;

      /* Manipulations to occur at each conntrack in this dirn. */
      struct ip6_conntrack_manip manip;
};

#ifdef __KERNEL__
#include <linux/list.h>
#include <linux/netfilter_ipv6/lockhelp.h>

/* Protects NAT hash tables, and NAT-private part of conntracks. */
DECLARE_RWLOCK_EXTERN(ip6_nat_lock);

/* Hashes for by-source and IP/protocol. */
```

```
struct ip6_nat_hash
{
      struct list_head list;

      /* conntrack we're embedded in: NULL if not in hash. */
      struct ip6_conntrack *conntrack;
};

/* The structure embedded in the conntrack structure. */
struct ip6_nat_info
{
      /* Set to zero when conntrack created: bitmask of maniptypes */
      int initialized;

      unsigned int num_manips;

      /* Manipulations to be done on this conntrack. */
      struct ip6_nat_info_manip manips[IP6_NAT_MAX_MANIPS];

      struct ip6_nat_hash bysource, byipsproto;

      /* Helper (NULL if none). */
      struct ip6_nat_helper *helper;

      struct ip6_nat_seq seq[IP6_CT_DIR_MAX];
};

/* Set up the info structure to map into this range. */
extern unsigned int ip6_nat_setup_info(struct ip6_conntrack *conntrack,
                              const struct ip6_nat_multi_range *mr,
                              unsigned int hooknum);

/* Is this tuple already taken? (not by us)*/
extern int ip6_nat_used_tuple(const struct ip6_conntrack_tuple *tuple,
                       const struct ip6_conntrack *ignored_conntrack);

/* Calculate relative checksum. */
extern u_int16_t ip6_nat_cheat_check(struct in6_addr oldvalinv,
                            struct in6_addr newval,
                            u_int16_t oldcheck);

extern u_int16_t ip6_int_nat_cheat_check(u_int32_t oldvalinv,
                            u_int32_t newval,
                            u_int16_t oldcheck);
#endif /*__KERNEL__*/
#endif
```

/NET/IPV6/NETFILTER/IP6_NAT_CORE.C

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: net/ipv4/netfilter/ip_nat_core.c
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

/* NAT for netfilter; shared with compatibility layer. */

/* (C) 1999-2001 Paul `Rusty' Russell
 * (C) 2002-2004 Netfilter Core Team <coreteam@netfilter.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */

#include <linux/module.h>
#include <linux/types.h>
#include <linux/timer.h>
#include <linux/skbuff.h>
#include <linux/netfilter_ipv6.h>
#include <linux/vmalloc.h>
#include <net/checksum.h>
#include <net/icmp.h>
#include <net/ipv6.h>
#include <net/tcp.h>  /* For tcp_prot in getorigdst */
#include <linux/icmpv6.h>
#include <linux/udp.h>

#define IPV6_HDR_LEN (sizeof(struct ipv6hdr))
#define ASSERT_READ_LOCK(x) MUST_BE_READ_LOCKED(&ip6_nat_lock)
#define ASSERT_WRITE_LOCK(x) MUST_BE_WRITE_LOCKED(&ip6_nat_lock)
```

```
#include <linux/netfilter_ipv6/ip6_conntrack.h>
#include <linux/netfilter_ipv6/ip6_conntrack_core.h>
#include <linux/netfilter_ipv6/ip6_conntrack_protocol.h>
#include <linux/netfilter_ipv6/ip6_nat.h>
#include <linux/netfilter_ipv6/ip6_nat_protocol.h>
#include <linux/netfilter_ipv6/ip6_nat_core.h>
#include <linux/netfilter_ipv6/ip6_nat_helper.h>
#include <linux/netfilter_ipv6/ip6_conntrack_helper.h>
#include <linux/netfilter_ipv4/listhelp.h>

#if 0
#define DEBUGP printk
#else
#define DEBUGP(format, args...)
#endif

DECLARE_RWLOCK(ip6_nat_lock);
DECLARE_RWLOCK_EXTERN(ip6_conntrack_lock);

/* Calculated at init based on memory size */
static unsigned int ip6_nat_htable_size;

static struct list_head *bysource;
static struct list_head *byipsproto;
LIST_HEAD(ip6_protos);
LIST_HEAD(ip6_helpers);

extern struct ip6_nat_protocol ip6_unknown_nat_protocol;

/* We keep extra hashes for each conntrack, for fast searching. */
static inline size_t
 hash_by_ipsproto(struct in6_addr src, struct in6_addr dst, u_int16_t
proto)
{
      /* Modified src and dst, to ensure we don't create two
           identical streams. */

      return (src.s6_addr32[0] + src.s6_addr32[1] + src.s6_addr32[2] +
src.s6_addr32[3] + dst.s6_addr32[0] + dst.s6_addr32[1] +
dst.s6_addr32[2] + dst.s6_addr32[3] + proto) % ip6_nat_htable_size;
}

static inline size_t
hash_by_src(const struct ip6_conntrack_manip *manip, u_int16_t proto)
{

      /* Original src, to ensure we map it consistently if poss. */
      return (manip->ip.s6_addr32[0] + manip->ip.s6_addr32[1] + manip-
>ip.s6_addr32[2] + manip->ip.s6_addr32[3] + manip->u.all + proto) %
ip6_nat_htable_size;
}

/* Noone using conntrack by the time this called. */
static void ip6_nat_cleanup_conntrack(struct ip6_conntrack *conn)
{

      struct ip6_nat_info *info = &conn->nat.info;
      unsigned int hs, hp;

 if (!info->initialized){
            return;
 }

      IP6_NF_ASSERT(info->bysource.conntrack);
      IP6_NF_ASSERT(info->byipsproto.conntrack);

      hs = hash_by_src(&conn->tuplehash[IP6_CT_DIR_ORIGINAL].tuple.src,
                       conn->tuplehash[IP6_CT_DIR_ORIGINAL]
                       .tuple.dst.protonum);

      hp                    =                    hash_by_ipsproto(conn-
>tuplehash[IP6_CT_DIR_REPLY].tuple.src.ip,
                            conn-
>tuplehash[IP6_CT_DIR_REPLY].tuple.dst.ip,
                            conn->tuplehash[IP6_CT_DIR_REPLY]
                            .tuple.dst.protonum);

      WRITE_LOCK(&ip6_nat_lock);
      LIST_DELETE(&bysource[hs], &info->bysource);
      LIST_DELETE(&byipsproto[hp], &info->byipsproto);
      WRITE_UNLOCK(&ip6_nat_lock);
}

/* We do checksum mangling, so if they were wrong before they're still
 * wrong.  Also works for incomplete packets (eg. ICMP dest
 * unreachables.) */



static  inline  int  cmp_proto(const  struct  ip6_nat_protocol  *i,  int
proto)
{
      return i->protonum == proto;
}

struct ip6_nat_protocol *
ip6_find_nat_proto(u_int16_t protonum)
{

      struct ip6_nat_protocol *i;
      MUST_BE_READ_LOCKED(&ip6_nat_lock);
      i = LIST_FIND(&ip6_protos, cmp_proto, struct ip6_nat_protocol *,
protonum);
 if (!i){
            i = &ip6_unknown_nat_protocol;
 }
      return i;
}

/* Is this tuple already taken? (not by us) */
int
ip6_nat_used_tuple(const struct ip6_conntrack_tuple *tuple,
              const struct ip6_conntrack *ignored_conntrack)
{
      /* Conntrack tracking doesn't keep track of outgoing tuples; only
         incoming ones.  NAT means they don't have a fixed mapping,
         so we invert the tuple and look for the incoming reply.
         We could keep a separate hash if this proves too slow. */

      struct ip6_conntrack_tuple reply;
      ip6_invert_tuplepr(&reply, tuple);

      return ip6_conntrack_tuple_taken(&reply, ignored_conntrack);
}

/* Does tuple + the source manip come within the range mr */
static int
in_range(const struct ip6_conntrack_tuple *tuple,
       const struct ip6_conntrack_manip *manip,
       const struct ip6_nat_multi_range *mr)
{
      struct   ip6_nat_protocol   *proto   =   ip6_find_nat_proto(tuple-
>dst.protonum);
      unsigned int i;
      struct ip6_conntrack_tuple newtuple = { *manip, tuple->dst };

      for (i = 0; i < mr->rangesize; i++) {
            /* If we are allowed to map IPs, then we must be in the
               range specified, otherwise we must be unchanged. */
            if (mr->range[i].flags & IP6_NAT_RANGE_MAP_IPS) {
                  if  (ntohl(newtuple.src.ip.s6_addr32[0])  <  ntohl(mr-
>range[i].min_ip.s6_addr32[0])
                      || (ntohl(newtuple.src.ip.s6_addr32[0])
                        > ntohl(mr->range[i].max_ip.s6_addr32[0]))){
                        continue;
                  }
            } else {

 if  ((newtuple.src.ip.s6_addr32[0]  !=  tuple->src.ip.s6_addr32[0])  ||
(newtuple.src.ip.s6_addr32[1]    !=    tuple->src.ip.s6_addr32[1])    ||
(newtuple.src.ip.s6_addr32[2]    !=    tuple->src.ip.s6_addr32[2])    ||
(newtuple.src.ip.s6_addr32[3] != tuple->src.ip.s6_addr32[3])){
                          continue;
                          }

            }

            if (!(mr->range[i].flags & IP6_NAT_RANGE_PROTO_SPECIFIED)
                || proto->in_range(&newtuple, IP6_NAT_MANIP_SRC,
                         &mr->range[i].min, &mr->range[i].max)){

                  return 1;
            }
      }
      return 0;
}

static inline int
src_cmp(const struct ip6_nat_hash *i,
      const struct ip6_conntrack_tuple *tuple,
      const struct ip6_nat_multi_range *mr)
```

```
{
      return                                              (i->conntrack-
>tuplehash[IP6_CT_DIR_ORIGINAL].tuple.dst.protonum
            == tuple->dst.protonum
            &&                                            (i->conntrack-
>tuplehash[IP6_CT_DIR_ORIGINAL].tuple.src.ip.s6_addr32[0]
                == tuple->src.ip.s6_addr32[0]
                &&                                         i->conntrack-
>tuplehash[IP6_CT_DIR_ORIGINAL].tuple.src.ip.s6_addr32[1]
                == tuple->src.ip.s6_addr32[1]
                &&                                         i->conntrack-
>tuplehash[IP6_CT_DIR_ORIGINAL].tuple.src.ip.s6_addr32[2]
                == tuple->src.ip.s6_addr32[2]
                &&                                         i->conntrack-
>tuplehash[IP6_CT_DIR_ORIGINAL].tuple.src.ip.s6_addr32[3]
                == tuple->src.ip.s6_addr32[3] )
            &&                                             i->conntrack-
>tuplehash[IP6_CT_DIR_ORIGINAL].tuple.src.u.all
            == tuple->src.u.all
            && in_range(tuple,
                      &i->conntrack->tuplehash[IP6_CT_DIR_ORIGINAL]
                      .tuple.src,
                      mr));
}

/* Only called for SRC manip */
static struct ip6_conntrack_manip *
find_appropriate_src(const struct ip6_conntrack_tuple *tuple,
                 const struct ip6_nat_multi_range *mr)
{
      unsigned int h = hash_by_src(&tuple->src, tuple->dst.protonum);
      struct ip6_nat_hash *i;
      MUST_BE_READ_LOCKED(&ip6_nat_lock);
      i  =  LIST_FIND(&bysource[h],  src_cmp,  struct  ip6_nat_hash  *,
tuple, mr);
      if (i){
            return                                        &i->conntrack-
>tuplehash[IP6_CT_DIR_ORIGINAL].tuple.src;
      }
      else{

      return NULL;
      }
}

/* Simple way to iterate through all. */
static inline int fake_cmp(const struct ip6_nat_hash *i,
                     struct   in6_addr   src,   struct   in6_addr   dst,
u_int16_t protonum,
                     unsigned int *score,
                     const struct ip6_conntrack *conntrack)
{
      /* Compare backwards: we're dealing with OUTGOING tuples, and
           inside the conntrack is the REPLY tuple.  Don't count this
           conntrack. */
      if (i->conntrack != conntrack
          &&                                              (i->conntrack-
>tuplehash[IP6_CT_DIR_REPLY].tuple.src.ip.s6_addr32[0]                ==
dst.s6_addr32[0]
            &&                                             i->conntrack-
>tuplehash[IP6_CT_DIR_REPLY].tuple.src.ip.s6_addr32[1]                ==
dst.s6_addr32[1]
            &&                                             i->conntrack-
>tuplehash[IP6_CT_DIR_REPLY].tuple.src.ip.s6_addr32[2]                ==
dst.s6_addr32[2]
            &&                                             i->conntrack-
>tuplehash[IP6_CT_DIR_REPLY].tuple.src.ip.s6_addr32[3]                ==
dst.s6_addr32[3])
        &&                                                (i->conntrack-
>tuplehash[IP6_CT_DIR_REPLY].tuple.dst.ip.s6_addr32[0]                ==
src.s6_addr32[0]
            &&                                             i->conntrack-
>tuplehash[IP6_CT_DIR_REPLY].tuple.dst.ip.s6_addr32[1]                ==
src.s6_addr32[1]
            &&                                             i->conntrack-
>tuplehash[IP6_CT_DIR_REPLY].tuple.dst.ip.s6_addr32[2]                ==
src.s6_addr32[2]
            &&                                             i->conntrack-
>tuplehash[IP6_CT_DIR_REPLY].tuple.dst.ip.s6_addr32[3]                ==
src.s6_addr32[3])
          &&                                              (i->conntrack-
>tuplehash[IP6_CT_DIR_REPLY].tuple.dst.protonum
            == protonum))
            (*score)++;
      return 0;
}

static inline unsigned int
count_maps(struct   in6_addr   src,   struct   in6_addr   dst,   u_int16_t
protonum,
         const struct ip6_conntrack *conntrack)
{
      unsigned int score = 0;
      unsigned int h;

      MUST_BE_READ_LOCKED(&ip6_nat_lock);
      h = hash_by_ipsproto(src, dst, protonum);
      LIST_FIND(&byipsproto[h], fake_cmp, struct ip6_nat_hash *,
                src, dst, protonum, &score, conntrack);

      return score;
}

/* For [FUTURE] fragmentation handling, we want the least-used
   src-ip/dst-ip/proto triple.  Fairness doesn't come into it.  Thus
   if the range specifies 1.2.3.4 ports 10000-10005 and 1.2.3.5 ports
   1-65535, we don't do pro-rata allocation based on ports; we choose
   the ip with the lowest src-ip/dst-ip/proto usage.

   If an allocation then fails (eg. all 6 ports used in the 1.2.3.4
   range), we eliminate that and try again.  This is not the most
   efficient approach, but if you're worried about that, don't hand us
   ranges you don't really have.  */
```

```
static struct ip6_nat_range *
find_best_ips_proto(struct ip6_conntrack_tuple *tuple,
                const struct ip6_nat_multi_range *mr,
                const struct ip6_conntrack *conntrack,
                unsigned int hooknum)
{
      unsigned int i;
      struct {
            const struct ip6_nat_range *range;
            unsigned int score;
            struct ip6_conntrack_tuple tuple;
      } best = { NULL,  0xFFFFFFFF };
      struct in6_addr *var_ipp, *other_ipp, saved_ip, orig_dstip;
      /*static unsigned int randomness;*/

      if (HOOK2MANIP(hooknum) == IP6_NAT_MANIP_SRC) {
            var_ipp = &tuple->src.ip;
            saved_ip = tuple->dst.ip;
            other_ipp = &tuple->dst.ip;
      } else {
            var_ipp = &tuple->dst.ip;
            saved_ip = tuple->src.ip;
            other_ipp = &tuple->src.ip;
      }
      /* Don't do do_extra_mangle unless necessary (overrides
           explicit socket bindings, for example) */
      orig_dstip = tuple->dst.ip;

      IP6_NF_ASSERT(mr->rangesize >= 1);
      for (i = 0; i < mr->rangesize; i++) {
            /* Host order */
        struct in6_addr minip, maxip;  /*, j;*/

            /* Don't do ranges which are already eliminated. */
            if (mr->range[i].flags & IP6_NAT_RANGE_FULL) {
                  continue;
            }

            if (mr->range[i].flags & IP6_NAT_RANGE_MAP_IPS) {
                  minip.s6_addr32[0]             =             ntohl(mr-
>range[i].min_ip.s6_addr32[0]);
                  minip.s6_addr32[1]             =             ntohl(mr-
>range[i].min_ip.s6_addr32[1]);
                  minip.s6_addr32[2]             =             ntohl(mr-
>range[i].min_ip.s6_addr32[2]);
                  minip.s6_addr32[3]             =             ntohl(mr-
>range[i].min_ip.s6_addr32[3]);

                  maxip.s6_addr32[0]             =             ntohl(mr-
>range[i].max_ip.s6_addr32[0]);
                  maxip.s6_addr32[1]             =             ntohl(mr-
>range[i].max_ip.s6_addr32[1]);
                  maxip.s6_addr32[2]             =             ntohl(mr-
>range[i].max_ip.s6_addr32[2]);
                  maxip.s6_addr32[3]             =             ntohl(mr-
>range[i].max_ip.s6_addr32[3]);
            }
            else {
              minip.s6_addr32[0]   =   maxip.s6_addr32[0]   =   var_ipp-
>s6_addr32[0];
              minip.s6_addr32[1]   =   maxip.s6_addr32[1]   =   var_ipp-
>s6_addr32[1];
              minip.s6_addr32[2]   =   maxip.s6_addr32[2]   =   var_ipp-
>s6_addr32[2];
              minip.s6_addr32[3]   =   maxip.s6_addr32[3]   =   var_ipp-
>s6_addr32[3];
            }


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - Not needed by our particular implementation. This function
was ported, but commented out because it was not tested, and was not
part of our implementation. Our basic NAT implementation did
 * not necessitate port translation or multiple IP address translation,
and so calculating
 * random IP addresses to use was not needed.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */

            /*
            randomness++;
            for (j = 0; j < maxip.s6_addr32[0] - minip.s6_addr32[0] +
1; j++) {
                  unsigned int score;

                  var_ipp->s6_addr32[0]  =  htonl(minip.s6_addr32[0]  +
(randomness + j)
                               %          (maxip.s6_addr32[0]          -
minip.s6_addr32[0] + 1));

                   Reset the other ip in case it was mangled by
                  do_extra_mangle last time.
                  other_ipp->s6_addr32[0] = saved_ip.s6_addr32[0];

#ifdef CONFIG_IP6_NF_NAT_LOCAL
                  if (hooknum == NF_IP6_LOCAL_OUT
                      &&            var_ipp->s6_addr32[0]            !=
orig_dstip.s6_addr32[0]
                      &&         !do_extra_mangle(var_ipp->s6_addr32[0],
other_ipp.s6_addr32[0])) {
                        DEBUGP("Range %u  %u:%u:%u:%u rt failed!\n",
                               i, NIP6(var_ipp->s6_addr32[0]));
                         Can't route?  This whole range part is
                          probably screwed, but keep trying
                          anyway.
                        continue;
                  }
#endif

                   Count how many others map onto this.
                  score = count_maps(tuple->src.ip.s6_addr32[0], tuple-
>dst.ip.s6_addr32[0],
                                 tuple->dst.protonum, conntrack);
                  if (score < best.score) {
                        Optimization: doesn't get any better than
                           this.
                        if (score == 0)
                              return (struct ip6_nat_range *)
                                    &mr->range[i];

                        best.score = score;
                        best.tuple = *tuple;
                        best.range = &mr->range[i];
                  }
            }
            */
            /* TB MP - END OF NAT CODE */



      }
      *tuple = best.tuple;

      /* Discard const. */
      return (struct ip6_nat_range *)best.range;
}

/* Fast version doesn't iterate through hash chains, but only handles
   common case of single IP address (null NAT, masquerade) */
static struct ip6_nat_range *
find_best_ips_proto_fast(struct ip6_conntrack_tuple *tuple,
                   const struct ip6_nat_multi_range *mr,
                   const struct ip6_conntrack *conntrack,
                   unsigned int hooknum)
{


      if (mr->rangesize != 1
          || (mr->range[0].flags & IP6_NAT_RANGE_FULL)
          || ((mr->range[0].flags & IP6_NAT_RANGE_MAP_IPS)
            &&       (mr->range[0].min_ip.s6_addr32[0]       !=       mr-
>range[0].max_ip.s6_addr32[0]
                ||      mr->range[0].min_ip.s6_addr32[1]      !=      mr-
>range[0].max_ip.s6_addr32[1]
                ||      mr->range[0].min_ip.s6_addr32[2]      !=      mr-
>range[0].max_ip.s6_addr32[2]
                ||      mr->range[0].min_ip.s6_addr32[3]      !=      mr-
>range[0].max_ip.s6_addr32[3])

            )){

      return find_best_ips_proto(tuple, mr, conntrack, hooknum);
      }
      if (mr->range[0].flags & IP6_NAT_RANGE_MAP_IPS) {
        if (HOOK2MANIP(hooknum) == IP6_NAT_MANIP_SRC){

                  tuple->src.ip = mr->range[0].min_ip;
 }
            else {
                  /* Only do extra mangle when required (breaks
                           socket binding) */
#ifdef CONFIG_IP6_NF_NAT_LOCAL
                  if      ((tuple->dst.ip.s6_addr32[0]       !=       mr-
>range[0].min_ip.s6_addr32[0]
                       ||     tuple->dst.ip.s6_addr32[1]      !=      mr-
>range[0].min_ip.s6_addr32[1]
                       ||     tuple->dst.ip.s6_addr32[2]      !=      mr-
>range[0].min_ip.s6_addr32[2]
                       ||     tuple->dst.ip.s6_addr32[3]      !=      mr-
>range[0].min_ip.s6_addr32[3])
                      && hooknum == NF_IP6_LOCAL_OUT){

                        return NULL;
                  }
#endif


                  tuple->dst.ip = mr->range[0].min_ip;
            }
      }

      /* Discard const. */


      return (struct ip6_nat_range *)&mr->range[0];
}

static int
get_unique_tuple(struct ip6_conntrack_tuple *tuple,
             const struct ip6_conntrack_tuple *orig_tuple,
             const struct ip6_nat_multi_range *mrr,
             struct ip6_conntrack *conntrack,
             unsigned int hooknum)
{


      struct ip6_nat_protocol *proto
            = ip6_find_nat_proto(orig_tuple->dst.protonum);
      struct ip6_nat_range *rptr;
      unsigned int i;
      int ret;

      /* We temporarily use flags for marking full parts, but we
         always clean up afterwards */
      struct ip6_nat_multi_range *mr = (void *)mrr;

      /* 1) If this srcip/proto/src-proto-part is currently mapped,
         and that same mapping gives a unique tuple within the given
         range, use that.

         This is only required for source (ie. NAT/masq) mappings.
         So far, we don't do local source mappings, so multiple
         manips not an issue.  */


      if (hooknum == NF_IP6_POST_ROUTING) {
            struct ip6_conntrack_manip *manip;


            manip = find_appropriate_src(orig_tuple, mr);


            if (manip) {
                  /* Apply same source manipulation. */
                  *tuple = ((struct ip6_conntrack_tuple)
                          { *manip, orig_tuple->dst });
                  DEBUGP("get_unique_tuple: Found current src map\n");


 if (!ip6_nat_used_tuple(tuple, conntrack)){
                        return 1;
 }
            }
      }

      /* 2) Select the least-used IP/proto combination in the given
         range.
      */
      *tuple = *orig_tuple;

      while  ((rptr  =  find_best_ips_proto_fast(tuple,  mr,  conntrack,
hooknum))
             != NULL) {
            DEBUGP("Found best for "); DUMP_TUPLE(tuple);
            /* 3) The per-protocol part of the manip is made to
               map into the range to make a unique tuple. */

            /* Only bother mapping if it's not already in range
               and unique */

            if ((!(rptr->flags & IP6_NAT_RANGE_PROTO_SPECIFIED)
                 || proto->in_range(tuple, HOOK2MANIP(hooknum),
                              &rptr->min, &rptr->max))
                && !ip6_nat_used_tuple(tuple, conntrack)) {
            ret = 1;

            goto clear_fulls;
            } else {

                  if (proto->unique_tuple(tuple, rptr,
                                    HOOK2MANIP(hooknum),
                                    conntrack)) {
                        /* Must be unique. */
                        IP6_NF_ASSERT(!ip6_nat_used_tuple(tuple,
                                                conntrack));

                        ret = 1;
                        goto clear_fulls;

                  }  else  if  (HOOK2MANIP(hooknum)  ==  IP6_NAT_MANIP_DST)
{
                        /* Try implicit source NAT; protocol
                                   may be able to play with ports to
                                   make it unique. */

                        struct ip6_nat_range r
                              = { IP6_NAT_RANGE_MAP_IPS,
                                  tuple->src.ip, tuple->src.ip,
                                  { 0 }, { 0 } };
                        DEBUGP("Trying implicit mapping\n");

                        if (proto->unique_tuple(tuple, &r,
                                         IP6_NAT_MANIP_SRC,
                                         conntrack)) {
                              /* Must be unique. */

                              IP6_NF_ASSERT(!ip6_nat_used_tuple
                                         (tuple, conntrack));

                              ret = 1;
                              goto clear_fulls;
                        }
                  }
                  DEBUGP("Protocol can't get unique tuple %u.\n",
                         hooknum);
            }

            /* Eliminate that from range, and try again. */

            rptr->flags |= IP6_NAT_RANGE_FULL;
            *tuple = *orig_tuple;
      }
      ret = 0;

 clear_fulls:
      /* Clear full flags. */
      IP6_NF_ASSERT(mr->rangesize >= 1);
      for (i = 0; i < mr->rangesize; i++)
            mr->range[i].flags &= ~IP6_NAT_RANGE_FULL;


      return ret;
}

static inline int
helper_cmp(const struct ip6_nat_helper *helper,
         const struct ip6_conntrack_tuple *tuple)
{
      return   ip6_ct_tuple_mask_cmp(tuple,   &helper->tuple,   &helper-
>mask);
}

/* Where to manip the reply packets (will be reverse manip). */
static unsigned int opposite_hook[NF_IP6_NUMHOOKS]
= { [NF_IP6_PRE_ROUTING] = NF_IP6_POST_ROUTING,
    [NF_IP6_POST_ROUTING] = NF_IP6_PRE_ROUTING,
#ifdef CONFIG_IP6_NF_NAT_LOCAL
    [NF_IP6_LOCAL_OUT] = NF_IP6_LOCAL_IN,
    [NF_IP6_LOCAL_IN] = NF_IP6_LOCAL_OUT,
#endif
```

```
};

unsigned int
ip6_nat_setup_info(struct ip6_conntrack *conntrack,
              const struct ip6_nat_multi_range *mr,
              unsigned int hooknum)
{
      struct ip6_conntrack_tuple new_tuple, inv_tuple, reply;
      struct ip6_conntrack_tuple orig_tp;
      struct ip6_nat_info *info = &conntrack->nat.info;
      int in_hashes = info->initialized;

      MUST_BE_WRITE_LOCKED(&ip6_nat_lock);

      IP6_NF_ASSERT(hooknum == NF_IP6_PRE_ROUTING
                 || hooknum == NF_IP6_POST_ROUTING
                 || hooknum == NF_IP6_LOCAL_OUT);

      IP6_NF_ASSERT(info->num_manips < IP6_NAT_MAX_MANIPS);
      IP6_NF_ASSERT(!(info->initialized & (1 << HOOK2MANIP(hooknum))));

      /* What we've got will look like inverse of reply. Normally
         this is what is in the conntrack, except for prior
         manipulations (future optimization: if num_manips == 0,
         orig_tp =
         conntrack->tuplehash[IP_CT_DIR_ORIGINAL].tuple) */

      ip6_invert_tuplepr(&orig_tp,
                   &conntrack->tuplehash[IP6_CT_DIR_REPLY].tuple);
#if 1
      {
      unsigned int i;
      DEBUGP("Hook %u (%s), ", hooknum,
             HOOK2MANIP(hooknum)==IP6_NAT_MANIP_SRC ? "SRC" : "DST");
      DUMP_TUPLE(&orig_tp);

      DEBUGP("Range %p: ", mr);
      for (i = 0; i < mr->rangesize; i++) {
            DEBUGP("%u:%s%s%s               %x:%x:%x:%x:%x:%x:%x:%x     -
%x:%x:%x:%x:%x:%x:%x:%x %u - %u\n",
                   i,
                   (mr->range[i].flags & IP6_NAT_RANGE_MAP_IPS)
                   ? " MAP_IPS" : "",
                   (mr->range[i].flags
                  & IP6_NAT_RANGE_PROTO_SPECIFIED)
                   ? " PROTO_SPECIFIED" : "",
                   (mr->range[i].flags & IP6_NAT_RANGE_FULL)
                   ? " FULL" : "",
                   NIP6(mr->range[i].min_ip),
                   NIP6(mr->range[i].max_ip),
                   mr->range[i].min.all,
                   mr->range[i].max.all);
      }
      }
#endif

      do {
            if (!get_unique_tuple(&new_tuple, &orig_tp, mr, conntrack,
                              hooknum)) {
                  DEBUGP("ip6_nat_setup_info:  Can't  get  unique  for
%p.\n",
                         conntrack);
                  return NF_DROP;
            }



#if 0


            DEBUGP("Hook %u (%s) %p\n", hooknum,
                   HOOK2MANIP(hooknum)==IP6_NAT_MANIP_SRC  ?  "SRC"  :
"DST",
                   conntrack);
            DEBUGP("Original: ");


            DUMP_TUPLE(&orig_tp);
            DEBUGP("New: ");


            DUMP_TUPLE(&new_tuple);
#endif

            /* We now have two tuples (SRCIP/SRCPT/DSTIP/DSTPT):
               the original (A/B/C/D') and the mangled one (E/F/G/H').
               We're only allowed to work with the SRC per-proto
               part, so we create inverses of both to start, then
               derive the other fields we need.  */

            /* Reply connection: simply invert the new tuple
                   (G/H/E/F') */

            ip6_invert_tuplepr(&reply, &new_tuple);

            /* Alter conntrack table so it recognizes replies.
                   If fail this race (reply tuple now used), repeat. */



      } while (!ip6_conntrack_alter_reply(conntrack, &reply));

      /* FIXME: We can simply used existing conntrack reply tuple
           here --RR */
      /* Create inverse of original: C/D/A/B' */

      ip6_invert_tuplepr(&inv_tuple, &orig_tp);

      /* Has source changed?. */

      if (!ip6_ct_tuple_src_equal(&new_tuple, &orig_tp)) {

            /* In this direction, a source manip. */
            info->manips[info->num_manips++] =
                ((struct ip6_nat_info_manip)
                 { IP6_CT_DIR_ORIGINAL, hooknum,
                   IP6_NAT_MANIP_SRC, new_tuple.src });



            IP6_NF_ASSERT(info->num_manips < IP6_NAT_MAX_MANIPS);


            /* In the reverse direction, a destination manip. */
            info->manips[info->num_manips++] =
                ((struct ip6_nat_info_manip)
                 { IP6_CT_DIR_REPLY, opposite_hook[hooknum],
                   IP6_NAT_MANIP_DST, orig_tp.src });
            IP6_NF_ASSERT(info->num_manips <= IP6_NAT_MAX_MANIPS);
      }

      /* Has destination changed? */
      if (!ip6_ct_tuple_dst_equal(&new_tuple, &orig_tp)) {
      /* In this direction, a destination manip */
            info->manips[info->num_manips++] =
                ((struct ip6_nat_info_manip)
                 { IP6_CT_DIR_ORIGINAL, hooknum,
                   IP6_NAT_MANIP_DST, reply.src });
            IP6_NF_ASSERT(info->num_manips < IP6_NAT_MAX_MANIPS);
            /* In the reverse direction, a source manip. */
            info->manips[info->num_manips++] =
                ((struct ip6_nat_info_manip)
                 { IP6_CT_DIR_REPLY, opposite_hook[hooknum],
                   IP6_NAT_MANIP_SRC, inv_tuple.src });

            IP6_NF_ASSERT(info->num_manips <= IP6_NAT_MAX_MANIPS);
      }

      /* If there's a helper, assign it; based on new tuple. */
 if (!conntrack->master){
            info->helper  =  LIST_FIND(&ip6_helpers,  helper_cmp,  struct
ip6_nat_helper *,
                               &reply);
 }
      /* It's done. */
      info->initialized |= (1 << HOOK2MANIP(hooknum));
      if (in_hashes) {
      IP6_NF_ASSERT(info->bysource.conntrack);
            ip6_replace_in_hashes(conntrack, info);
      } else {
            ip6_place_in_hashes(conntrack, info);
      }
      return NF_ACCEPT;
}

void ip6_replace_in_hashes(struct ip6_conntrack *conntrack,
                   struct ip6_nat_info *info)
{

      /* Source has changed, so replace in hashes. */
      unsigned int srchash
```

```
            = hash_by_src(&conntrack->tuplehash[IP6_CT_DIR_ORIGINAL]
                        .tuple.src,
                        conntrack->tuplehash[IP6_CT_DIR_ORIGINAL]
                        .tuple.dst.protonum);
      /* We place packet as seen OUTGOUNG in byips_proto hash
           (ie. reverse dst and src of reply packet. */
      unsigned int ipsprotohash
            = hash_by_ipsproto(conntrack->tuplehash[IP6_CT_DIR_REPLY]
                           .tuple.dst.ip,
                           conntrack->tuplehash[IP6_CT_DIR_REPLY]
                           .tuple.src.ip,
                           conntrack->tuplehash[IP6_CT_DIR_REPLY]
                           .tuple.dst.protonum);

      IP6_NF_ASSERT(info->bysource.conntrack == conntrack);
      MUST_BE_WRITE_LOCKED(&ip6_nat_lock);
      list_del(&info->bysource.list);
      list_del(&info->byipsproto.list);
      list_prepend(&bysource[srchash], &info->bysource);
      list_prepend(&byipsproto[ipsprotohash], &info->byipsproto);
}

void ip6_place_in_hashes(struct ip6_conntrack *conntrack,
                 struct ip6_nat_info *info)
{
      unsigned int srchash
            = hash_by_src(&conntrack->tuplehash[IP6_CT_DIR_ORIGINAL]
                        .tuple.src,
                        conntrack->tuplehash[IP6_CT_DIR_ORIGINAL]
                        .tuple.dst.protonum);
      /* We place packet as seen OUTGOUNG in byips_proto hash
           (ie. reverse dst and src of reply packet. */
      unsigned int ipsprotohash
            = hash_by_ipsproto(conntrack->tuplehash[IP6_CT_DIR_REPLY]
                           .tuple.dst.ip,
                           conntrack->tuplehash[IP6_CT_DIR_REPLY]
                           .tuple.src.ip,
                           conntrack->tuplehash[IP6_CT_DIR_REPLY]
                           .tuple.dst.protonum);

      IP6_NF_ASSERT(!info->bysource.conntrack);
      MUST_BE_WRITE_LOCKED(&ip6_nat_lock);
      info->byipsproto.conntrack = conntrack;
      info->bysource.conntrack = conntrack;
      list_prepend(&bysource[srchash], &info->bysource);
      list_prepend(&byipsproto[ipsprotohash], &info->byipsproto);
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - The manip_pkt function necessitated some changes due to the
introduction of
 * a pseudo-header to ICMPv6 header checksum calculation and the
removal of the IP header
```

```
 * checksum.  In IPv4, the checksum of the ICMP packet was calculated
first, then the IP
 * addresses were translated and an IP checksum calculated.  IPv6 no
longer has a checksum
 * in the header, so those checksum calculations were removed.  Since
the translated IP addresses need to be part of the ICMP pseudo-header,
the order of operations in this function
 * was switched so that the IP addresses are translated first, then the
upper layer header gets
 * manipulated.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */

/* Returns true if succeeded. */
static int
manip_pkt(u_int16_t proto,
        struct sk_buff **pskb,
        unsigned int ipv6hdroff,
        const struct ip6_conntrack_manip *manip,
        enum ip6_nat_manip_type maniptype)
{
      struct ipv6hdr *ipv6h;
      (*pskb)->nfcache |= NFC_ALTERED;
      if (!skb_ip6_make_writable(pskb, ipv6hdroff+sizeof(ipv6h))){
            return 0;
 }

      ipv6h = (void *)(*pskb)->data + ipv6hdroff;
      if (maniptype == IP6_NAT_MANIP_SRC) {


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - IPv6 headers do not have checksums, therefore these checksum
calculations are
 * not necessary. The IPv4 code that was here:
 *
 * iph->check = ip_nat_cheat_check(~iph->saddr, manip->ip,
 *                            iph->check);
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */

        ipv6h->saddr = manip->ip;

      } else {

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - IPv6 headers do not have checksums, therefore these checksum
calculations are
 * not necessary. The IPv4 code that was here:
 *
 * iph->check = ip_nat_cheat_check(~iph->saddr, manip->ip,
 *                            iph->check);
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */
```

```
          ipv6h->daddr = manip->ip;
      }


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - This part manipulates the upper layer header information
using the new IP addresses.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */

      /* Manipulate protcol part. */
      if (!ip6_find_nat_proto(proto)->manip_pkt(pskb,
                              ipv6hdroff + IPV6_HDR_LEN,
                          manip, maniptype)){
            return 0;
      }
      ipv6h = (void *)(*pskb)->data + ipv6hdroff;
      return 1;
}

static inline int exp_for_packet(struct ip6_conntrack_expect *exp,
                           struct sk_buff *skb,
                         unsigned int dataoff)
{
      struct ip6_conntrack_protocol *proto;
      int ret = 1;

      MUST_BE_READ_LOCKED(&ip6_conntrack_lock);
      proto = __ip6_ct_find_proto(skb->nh.ipv6h->nexthdr);
      if (proto->exp_matches_pkt)
            ret = proto->exp_matches_pkt(exp, skb, dataoff);

      return ret;
}

/* Do packet manipulations according to binding. */
unsigned int
ip6_do_bindings(struct ip6_conntrack *ct,
          enum ip6_conntrack_info ctinfo,
          struct ip6_nat_info *info,
          unsigned int hooknum,
          struct sk_buff **pskb,
          unsigned int dataoff)
{
      unsigned int i;
      struct ip6_nat_helper *helper;
      enum ip6_conntrack_dir dir = CTINFO2DIR(ctinfo);
      int proto = (*pskb)->nh.ipv6h->nexthdr;

      /* Need nat lock to protect against modification, but neither
         conntrack (referenced) and helper (deleted with
         synchronize_bh()) can vanish. */

      READ_LOCK(&ip6_nat_lock);

      for (i = 0; i < info->num_manips; i++) {
            if (info->manips[i].direction == dir
                && info->manips[i].hooknum == hooknum) {
                  DEBUGP("Mangling %p: %s to %x:%x:%x:%x:%x:%x:%x:%x
%u\n",
                         *pskb,
                         info->manips[i].maniptype == IP6_NAT_MANIP_SRC
                         ? "SRC" : "DST",
                         NIP6(info->manips[i].manip.ip),
                         htons(info->manips[i].manip.u.all));
                  if (!manip_pkt(proto, pskb, 0,
                               &info->manips[i].manip,
                               info->manips[i].maniptype)) {

                        READ_UNLOCK(&ip6_nat_lock);

                        return NF_DROP;
                  }
            }
      }
      helper = info->helper;
      READ_UNLOCK(&ip6_nat_lock);
      if (helper) {
            struct ip6_conntrack_expect *exp = NULL;
            struct list_head *cur_item;
            int ret = NF_ACCEPT;
            int helper_called = 0;
            DEBUGP("ip6_do_bindings: helper existing for (%p)\n", ct);
            /* Always defragged for helpers */
            IP6_NF_ASSERT(!((*pskb)->nh.ipv6h->frag_off
                         & htons(IP6_MF|IP6_OFFSET)));
            /* Have to grab read lock before sibling_list traversal */
            READ_LOCK(&ip6_conntrack_lock);
            list_for_each(cur_item, &ct->sibling_list) {
                  exp = list_entry(cur_item, struct
ip6_conntrack_expect,
                               expected_list);
            /* if this expectation is already established, skip */
 if (exp->sibling){
                        continue;
 }
                  if (exp_for_packet(exp, *pskb, dataoff)) {
                        /* FIXME: May be true multiple times in the
                         * case of UDP!! */
                        DEBUGP("calling nat helper (exp=%p) for
packet\n", exp);
                        ret = helper->help(ct, exp, info, ctinfo,
                                 hooknum, pskb);
                        if (ret != NF_ACCEPT) {
                              READ_UNLOCK(&ip6_conntrack_lock);
                              return ret;
                        }
                  helper_called = 1;
                  }
            }
            /* Helper might want to manip the packet even when there is
no
             * matching expectation for this packet */
            if (!helper_called && helper->flags &
IP6_NAT_HELPER_F_ALWAYS) {
                  DEBUGP("calling nat helper for packet without
expectation\n");
                  ret = helper->help(ct, NULL, info, ctinfo,
                                 hooknum, pskb);
                  if (ret != NF_ACCEPT) {
                        READ_UNLOCK(&ip6_conntrack_lock);
                        return ret;
                  }
            }
            READ_UNLOCK(&ip6_conntrack_lock);

            /* Adjust sequence number only once per packet
             * (helper is called at all hooks) */

            if (proto == IPPROTO_TCP
                && (hooknum == NF_IP6_POST_ROUTING
                  || hooknum == NF_IP6_LOCAL_IN)) {
                  DEBUGP("ip6_nat_core: adjusting sequence number\n");
                  /* future: put this in a l4-proto specific function,
                   * and call this function here. */
 if (!ip6_nat_seq_adjust(pskb, ct, ctinfo)){
                        ret = NF_DROP;
 }
            }
            return ret;
      } else {
            return NF_ACCEPT;
      }
      /* not reached */
}

int
icmpv6_reply_translation(struct sk_buff **pskb,
                   struct ip6_conntrack *conntrack,
                   unsigned int hooknum,
                   int dir)
{
      struct {
            struct icmp6hdr icmp;
            struct ipv6hdr ip;
      } *inside;
      unsigned int i;
      struct ip6_nat_info *info = &conntrack->nat.info;
      int hdrlen;
      if (!skb_ip6_make_writable(pskb, IPV6_HDR_LEN + sizeof(*inside)))
            return 0;
      inside = (void *)(*pskb)->data + IPV6_HDR_LEN;

      /* We're actually going to mangle it beyond trivial checksum
         adjustment, so make sure the current checksum is correct. */

      if ((*pskb)->ip_summed != CHECKSUM_UNNECESSARY) {
            hdrlen = IPV6_HDR_LEN;

}
```

```
      /* Must be RELATED */
      IP6_NF_ASSERT((*pskb)->nfct
                 - (struct ip6_conntrack *)(*pskb)->nfct->master
                 == IP6_CT_RELATED
                 || (*pskb)->nfct
                 - (struct ip6_conntrack *)(*pskb)->nfct->master
                 == IP6_CT_RELATED+IP6_CT_IS_REPLY);
      /* Redirects on non-null nats must be dropped, else they'll
           start talking to each other without our translation, and be
           confused... --RR */


      DEBUGP("icmpv6_reply_translation:  translating  error  %p  hook  %u
dir %s\n",
             *pskb,  hooknum,  dir  ==  IP6_CT_DIR_ORIGINAL  ?  "ORIG"  :
"REPLY");
      /* Note: May not be from a NAT'd host, but probably safest to
         do translation always as if it came from the host itself
         (even though a "host unreachable" coming from the host
         itself is a bit weird).

         More explanation: some people use NAT for anonymizing.
         Also, CERT recommends dropping all packets from private IP
         addresses (although ICMP errors from internal links with
         such addresses are not too uncommon, as Alan Cox points
         out) */

      READ_LOCK(&ip6_nat_lock);
      for (i = 0; i < info->num_manips; i++) {
            DEBUGP("icmpv6_reply: manip %u dir %s hook %u\n",
                   i,  info->manips[i].direction  ==  IP6_CT_DIR_ORIGINAL
?
                   "ORIG" : "REPLY", info->manips[i].hooknum);

            if (info->manips[i].direction != dir){
              continue;
            }

            /* Mapping the inner packet is just like a normal
               packet, except it was never src/dst reversed, so
               where we would normally apply a dst manip, we apply
               a src, and vice versa. */

            if (info->manips[i].hooknum == hooknum) {
                  DEBUGP("icmpv6_reply:        inner        %s        ->
%x:%x:%x:%x:%x:%x:%x:%x %u\n",
                         info->manips[i].maniptype == IP6_NAT_MANIP_SRC
                         ? "DST" : "SRC",
                         NIP6(info->manips[i].manip.ip),
                         ntohs(info->manips[i].manip.u.udp.port));

                  if (!manip_pkt(inside->ip.nexthdr, pskb,
                               IPV6_HDR_LEN
                               + sizeof(inside->icmp),
                               &info->manips[i].manip,
                               !info->manips[i].maniptype)){
```

```
goto unlock_fail;
                  }
                  /* Outer packet needs to have IP header NATe}d like
                         it's a reply. */

                  /* Use mapping to map outer packet: 0 give no
                           per-proto mapping */
                  DEBUGP("icmpv6_reply:        outer        %s        ->
%x:%x:%x:%x:%x:%x:%x:%x \n",
                         info->manips[i].maniptype == IP6_NAT_MANIP_SRC
                         ? "SRC" : "DST",
                         NIP6(info->manips[i].manip.ip));
                  if (!manip_pkt(0, pskb, 0,
                               &info->manips[i].manip,
                               info->manips[i].maniptype)){


goto unlock_fail;
                  }
            }
      }
      READ_UNLOCK(&ip6_nat_lock);

      hdrlen = IPV6_HDR_LEN;

      inside = (void *)(*pskb)->data + IPV6_HDR_LEN;

            struct in6_addr *saddrtmp, *daddrtmp;
            struct sk_buff *skb = *pskb;

      saddrtmp = &skb->nh.ipv6h->saddr;
        daddrtmp = &skb->nh.ipv6h->daddr;

      inside->icmp.icmp6_cksum = 0;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - Here we use the csum_ipv6_magic and csum_partial functions
to calculate the
 * ICMPv6 header checksum.  csum_partial determines the checksum for
just the ICMPv6 header
 * but does not flip the bits at the end.  This is then folded into the
pseudo-header checksum
 * calculation done by csum_ipv6_magic, which then yields a proper
checksum for the entire
 * ICMPv6 header and pseudo-header combination.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */


inside->icmp.icmp6_cksum = csum_ipv6_magic(saddrtmp, daddrtmp, (*pskb)-
>len - sizeof(struct ipv6hdr), IPPROTO_ICMPV6,
 csum_partial((char   *)&inside->icmp,   (*pskb)->len   -   sizeof(struct
ipv6hdr), 0));

      return 1;
 unlock_fail:
      READ_UNLOCK(&ip6_nat_lock);
      return 0;
}

int __init ip6_nat_init(void)

{
      size_t i;
      /* Leave them the same for the moment. */
      ip6_nat_htable_size = ip6_conntrack_htable_size;

      /* One vmalloc for both hash tables */
      bysource       =      vmalloc(sizeof(struct      list_head)      *
ip6_nat_htable_size*2);
      if (!bysource) {
            return -ENOMEM;
      }
      byipsproto = bysource + ip6_nat_htable_size;

      /* Sew in builtin protocols. */
      WRITE_LOCK(&ip6_nat_lock);
      list_append(&ip6_protos, &ip6_nat_protocol_tcp);
      list_append(&ip6_protos, &ip6_nat_protocol_udp);
      list_append(&ip6_protos, &ip6_nat_protocol_icmp);
      WRITE_UNLOCK(&ip6_nat_lock);

      for (i = 0; i < ip6_nat_htable_size; i++) {
            INIT_LIST_HEAD(&bysource[i]);
            INIT_LIST_HEAD(&byipsproto[i]);
      }

      /* FIXME: Man, this is a hack.  <SIGH> */
      IP6_NF_ASSERT(ip6_conntrack_destroyed == NULL);
      ip6_conntrack_destroyed = &ip6_nat_cleanup_conntrack;

      return 0;
}

/* Clear NAT section of all conntracks, in case we're loaded again. */
static int clean_nat(const struct ip6_conntrack *i, void *data)
{
      memset((void *)&i->nat, 0, sizeof(i->nat));
      return 0;
}

/* Not __exit: called from ip6_nat_standalone.c:init_or_cleanup() --RR
*/
void ip6_nat_cleanup(void)
{
      ip6_ct_selective_cleanup(&clean_nat, NULL);
      ip6_conntrack_destroyed = NULL;
      vfree(bysource);
}
```

/INCLUDE/LINUX/NETFILTER_IPV6/IP6_NAT_CORE.H

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: include/linux/ip_nat_core.h
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

#ifndef _IP6_NAT_CORE_H
#define _IP6_NAT_CORE_H
#include <linux/list.h>
#include <linux/netfilter_ipv6/ip6_conntrack.h>

/* This header used to share core functionality between the standalone
   NAT  module,  and  the  compatibility  layer's  use  of  NAT  for
masquerading. */
extern int ip6_nat_init(void);
extern void ip6_nat_cleanup(void);

extern unsigned int ip6_do_bindings(struct ip6_conntrack *ct,
                        enum ip6_conntrack_info conntrackinfo,
                        struct ip6_nat_info *info,
                        unsigned int hooknum,
                        struct sk_buff **pskb,
                        unsigned int dataoff);

extern struct list_head ip6_protos;

extern int icmpv6_reply_translation(struct sk_buff **pskb,
                          struct ip6_conntrack *conntrack,
                          unsigned int hooknum,
                          int dir);

extern void ip6_replace_in_hashes(struct ip6_conntrack *conntrack,
                        struct ip6_nat_info *info);
```

```
extern void ip6_place_in_hashes(struct ip6_conntrack *conntrack,
                      struct ip6_nat_info *info);

/* Built-in protocols. */
extern struct ip6_nat_protocol ip6_nat_protocol_tcp;
extern struct ip6_nat_protocol ip6_nat_protocol_udp;
extern struct ip6_nat_protocol ip6_nat_protocol_icmp;
#endif /* _IP6_NAT_CORE_H */
```

/NET/IPV6/NETFILTER/IP6_NAT_HELPER.C

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: net/ipv4/netfilter/ip_nat_helper.c
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * This file was ported, yet due to the scope of this thesis, no helper
 * files were used. This file was not tested.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

/* ip_nat_helper.c - generic support functions for NAT helpers
 *
 * (C) 2000-2002 Harald Welte <laforge@netfilter.org>
 * (C) 2003-2004 Netfilter Core Team <coreteam@netfilter.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 *
 *    14 Jan 2002 Harald Welte <laforge@gnumonks.org>:
 *          - add support for SACK adjustment
 *    14 Mar 2002 Harald Welte <laforge@gnumonks.org>:
 *          - merge SACK support into newnat API
 *    16 Aug 2002 Brian J. Murrell <netfilter@interlinx.bc.ca>:
 *          - make ip_nat_resize_packet more generic (TCP and UDP)
 *          - add ip_nat_mangle_udp_packet
 */

#include <linux/config.h>
#include <linux/module.h>
#include <linux/kmod.h>
#include <linux/types.h>
#include <linux/timer.h>
#include <linux/skbuff.h>
```

```
#include <linux/netfilter_ipv6.h>
#include <net/checksum.h>
#include <net/icmp.h>
#include <net/ipv6.h>
#include <net/tcp.h>
#include <net/udp.h>

#define IPV6_HDR_LEN    (sizeof(struct ipv6hdr))
#define ASSERT_READ_LOCK(x) MUST_BE_READ_LOCKED(&ip6_nat_lock)
#define ASSERT_WRITE_LOCK(x) MUST_BE_WRITE_LOCKED(&ip6_nat_lock)

#include <linux/netfilter_ipv6/ip6_conntrack.h>
#include <linux/netfilter_ipv6/ip6_conntrack_helper.h>
#include <linux/netfilter_ipv6/ip6_nat.h>
#include <linux/netfilter_ipv6/ip6_nat_protocol.h>
#include <linux/netfilter_ipv6/ip6_nat_core.h>
#include <linux/netfilter_ipv6/ip6_nat_helper.h>
#include <linux/netfilter_ipv4/listhelp.h>

#if 0
#define DEBUGP printk
#define DUMP_OFFSET(x)  printk("offset_before=%d,       offset_after=%d,
correction_pos=%u\n",      x->offset_before,     x->offset_after,     x-
>correction_pos);
#else
#define DEBUGP(format, args...)
#define DUMP_OFFSET(x)
#endif

DECLARE_LOCK(ip6_nat_seqofs_lock);

/* Setup TCP sequence correction given this change at this sequence */
static inline void
adjust_tcp_sequence(u32 seq,
                int sizediff,
                struct ip6_conntrack *ct,
                enum ip6_conntrack_info ctinfo)
{
      int dir;
      struct ip6_nat_seq *this_way, *other_way;

      DEBUGP("ip6_nat_resize_packet: old_size = %u, new_size = %u\n",
            (*skb)->len, new_size);

      dir = CTINFO2DIR(ctinfo);

      this_way = &ct->nat.info.seq[dir];
      other_way = &ct->nat.info.seq[!dir];

      DEBUGP("ip6_nat_resize_packet: Seq_offset before: ");
      DUMP_OFFSET(this_way);

      LOCK_BH(&ip6_nat_seqofs_lock);

      /* SYN adjust. If it's uninitialized, of this is after last
       * correction, record it: we don't handle more than one
       * adjustment in the window, but do deal with common case of a
       * retransmit */
      if (this_way->offset_before == this_way->offset_after
          || before(this_way->correction_pos, seq)) {
                this_way->correction_pos = seq;
                this_way->offset_before = this_way->offset_after;
                this_way->offset_after += sizediff;
      }
      UNLOCK_BH(&ip6_nat_seqofs_lock);

      DEBUGP("ip6_nat_resize_packet: Seq_offset after: ");
      DUMP_OFFSET(this_way);
}
/* Frobs data inside this packet, which is linear. */
static void mangle_contents(struct sk_buff *skb,
                      unsigned int dataoff,
                      unsigned int match_offset,
                      unsigned int match_len,
                      const char *rep_buffer,
                      unsigned int rep_len)
{
      unsigned char *data;

      BUG_ON(skb_is_nonlinear(skb));
      data = (unsigned char *)skb->nh.ipv6h + dataoff;

      /* move post-replacement */
      memmove(data + match_offset + rep_len,
            data + match_offset + match_len,
            skb->tail - (data + match_offset + match_len));

      /* insert data from buffer */
      memcpy(data + match_offset, rep_buffer, rep_len);

      /* update skb info */
      if (rep_len > match_len) {
            DEBUGP("ip6_nat_mangle_packet: Extending packet by "
                  "%u from %u bytes\n", rep_len - match_len,
                   skb->len);
            skb_put(skb, rep_len - match_len);
      } else {
            DEBUGP("ip6_nat_mangle_packet: Shrinking packet from "
                  "%u from %u bytes\n", match_len - rep_len,
                   skb->len);
            __skb_trim(skb, skb->len + rep_len - match_len);
      }



/* Unusual, but possible case. */
static int enlarge_skb(struct sk_buff **pskb, unsigned int extra)
{
      struct sk_buff *nskb;

      if ((*pskb)->len + extra > 65535)
            return 0;
      nskb   =   skb_copy_expand(*pskb,   skb_headroom(*pskb),   extra,
GFP_ATOMIC);
      if (!nskb)
            return 0;

      /* Transfer socket to new skb. */
      if ((*pskb)->sk)
            skb_set_owner_w(nskb, (*pskb)->sk);
#ifdef CONFIG_NETFILTER_DEBUG
      nskb->nf_debug = (*pskb)->nf_debug;
#endif
      kfree_skb(*pskb);
      *pskb = nskb;
      return 1;
}

/* Generic function for mangling variable-length address changes inside
 * NATed TCP connections (like the PORT XXX,XXX,XXX,XXX,XXX,XXX
 * command in FTP).
 *
 *  Takes  care  about  all  the  nasty  sequence  number  changes,
checksumming,
 * skb enlargement, ...
 *
 * */

static __inline__ u16 tcp_v6_check(struct tcphdr *th, int len,
                           struct in6_addr *saddr,
                           struct in6_addr *daddr,
                           unsigned long base)
{
      return csum_ipv6_magic(saddr, daddr, len, IPPROTO_TCP, base);
}

int
ip6_nat_mangle_tcp_packet(struct sk_buff **pskb,
                   struct ip6_conntrack *ct,
                   enum ip6_conntrack_info ctinfo,
                   unsigned int match_offset,
                   unsigned int match_len,
                   const char *rep_buffer,
                   unsigned int rep_len)
{
      struct ipv6hdr *ipv6h;
      struct tcphdr *tcph;

      int datalen;

      if (!skb_ip6_make_writable(pskb, (*pskb)->len))
            return 0;

      if (rep_len > match_len
          && rep_len - match_len > skb_tailroom(*pskb)
          && !enlarge_skb(pskb, rep_len - match_len))
            return 0;

      SKB_LINEAR_ASSERT(*pskb);
      ipv6h = (*pskb)->nh.ipv6h;
      tcph = (void *)ipv6h + IPV6_HDR_LEN;

      mangle_contents(*pskb, IPV6_HDR_LEN + tcph->doff*4,
                  match_offset, match_len, rep_buffer, rep_len);

      datalen = (*pskb)->len - IPV6_HDR_LEN;

      tcph->check = 0;
      tcph->check = tcp_v6_check(tcph, datalen, &ipv6h->saddr, &ipv6h-
>daddr,
                           csum_partial((char *)tcph, datalen, 0));

      adjust_tcp_sequence(ntohl(tcph->seq),
                      (int)rep_len - (int)match_len,
                      ct, ctinfo);
      return 1;
}

/* Generic function for mangling variable-length address changes inside
 * NATed UDP connections (like the CONNECT DATA XXXXX MESG XXXXX INDEX
XXXXX
 * command in the Amanda protocol)
 *
 * Takes care about all the nasty sequence number changes,
checksumming,
 * skb enlargement, ...
 *
 * XXX - This function could be merged with ip_nat_mangle_tcp_packet
which
 *       should be fairly easy to do.
 */
int
ip6_nat_mangle_udp_packet(struct sk_buff **pskb,
                   struct ip6_conntrack *ct,
                   enum ip6_conntrack_info ctinfo,
                   unsigned int match_offset,
                   unsigned int match_len,
                   const char *rep_buffer,
                   unsigned int rep_len)
{
      struct ipv6hdr *ipv6h;
      struct udphdr *udph;

      /* UDP helpers might accidentally mangle the wrong packet */
      ipv6h = (*pskb)->nh.ipv6h;
      if ((*pskb)->len < IPV6_HDR_LEN + sizeof(*udph) +
                            match_offset + match_len)
            return 0;

      if (!skb_ip6_make_writable(pskb, (*pskb)->len))
            return 0;

      if (rep_len > match_len
          && rep_len - match_len > skb_tailroom(*pskb)
          && !enlarge_skb(pskb, rep_len - match_len))
```

```
            return 0;

      ipv6h = (*pskb)->nh.ipv6h;
      udph = (void *)ipv6h + IPV6_HDR_LEN;
      mangle_contents(*pskb, IPV6_HDR_LEN + sizeof(*udph),
                  match_offset, match_len, rep_buffer, rep_len);

      /* update the length of the UDP packet */
      udph->len = htons((*pskb)->len - IPV6_HDR_LEN);

      /* fix udp checksum if udp checksum was previously calculated */
      if (udph->check) {
            int datalen = (*pskb)->len - IPV6_HDR_LEN;
            udph->check = 0;
            udph->check = csum_ipv6_magic(&ipv6h->saddr, &ipv6h->daddr,
                                            datalen, IPPROTO_UDP,
                                            csum_partial((char *)udph,
                                                         datalen, 0));
      }

      return 1;
}

/* Adjust one found SACK option including checksum correction */
static void
sack_adjust(struct sk_buff *skb,
          struct tcphdr *tcph,
          unsigned int sackoff,
          unsigned int sackend,
          struct ip6_nat_seq *natseq)
{
      while (sackoff < sackend) {
            struct tcp_sack_block *sack;
            u_int32_t new_start_seq, new_end_seq;

            sack = (void *)skb->data + sackoff;
            if (after(ntohl(sack->start_seq) - natseq->offset_before,
                    natseq->correction_pos))
                  new_start_seq = ntohl(sack->start_seq)
                              - natseq->offset_after;
            else
                  new_start_seq = ntohl(sack->start_seq)
                              - natseq->offset_before;
            new_start_seq = htonl(new_start_seq);

            if (after(ntohl(sack->end_seq) - natseq->offset_before,
                    natseq->correction_pos))
                  new_end_seq = ntohl(sack->end_seq)
                              - natseq->offset_after;
            else
                  new_end_seq = ntohl(sack->end_seq)
                              - natseq->offset_before;
            new_end_seq = htonl(new_end_seq);

            DEBUGP("sack_adjust: start_seq: %d->%d, end_seq: %d->%d\n",
                  ntohl(sack->start_seq), new_start_seq,
                  ntohl(sack->end_seq), new_end_seq);
            tcph->check =
                  ip6_int_nat_cheat_check(~sack->start_seq,
new_start_seq,
                                 ip6_int_nat_cheat_check(~sack-
>end_seq,
                                                new_end_seq,
                                                tcph->check));
            sack->start_seq = new_start_seq;
            sack->end_seq = new_end_seq;
            sackoff += sizeof(*sack);
      }
}

/* TCP SACK sequence number adjustment */
static inline unsigned int
ip6_nat_sack_adjust(struct sk_buff **pskb,
               struct tcphdr *tcph,
               struct ip6_conntrack *ct,
               enum ip6_conntrack_info ctinfo)
{
      unsigned int dir, optoff, optend;

      optoff = IPV6_HDR_LEN + sizeof(struct tcphdr);
      optend = IPV6_HDR_LEN + tcph->doff*4;

      if (!skb_ip6_make_writable(pskb, optend))
            return 0;

      dir = CTINFO2DIR(ctinfo);

      while (optoff < optend) {
            /* Usually: option, length. */
            unsigned char *op = (*pskb)->data + optoff;

            switch (op[0]) {
            case TCPOPT_EOL:
                  return 1;
            case TCPOPT_NOP:
                  optoff++;
                  continue;
            default:
                  /* no partial options */
                  if (optoff + 1 == optend
                      || optoff + op[1] > optend
                      || op[1] < 2)
                        return 0;
                  if (op[0] == TCPOPT_SACK
                      && op[1] >= 2+TCPOLEN_SACK_PERBLOCK
                      && ((op[1] - 2) % TCPOLEN_SACK_PERBLOCK) == 0)
                        sack_adjust(*pskb, tcph, optoff+2,
                                  optoff+op[1],
                                  &ct->nat.info.seq[!dir]);
                  optoff += op[1];
            }
      }
      return 1;
```

```
}

/* TCP sequence number adjustment.  Returns true or false.  */
int
ip6_nat_seq_adjust(struct sk_buff **pskb,
              struct ip6_conntrack *ct,
              enum ip6_conntrack_info ctinfo)
{
      struct tcphdr *tcph;
      int dir, newseq, newack;
      struct ip6_nat_seq *this_way, *other_way;

      dir = CTINFO2DIR(ctinfo);

      this_way = &ct->nat.info.seq[dir];
      other_way = &ct->nat.info.seq[!dir];

      /* No adjustments to make?  Very common case. */
      if (!this_way->offset_before && !this_way->offset_after
          && !other_way->offset_before && !other_way->offset_after)
            return 1;

      if (!skb_ip6_make_writable(pskb, IPV6_HDR_LEN+sizeof(*tcph)))
            return 0;

      tcph = (void *)(*pskb)->data + IPV6_HDR_LEN;
      if (after(ntohl(tcph->seq), this_way->correction_pos))
            newseq = ntohl(tcph->seq) + this_way->offset_after;
      else
            newseq = ntohl(tcph->seq) + this_way->offset_before;
      newseq = htonl(newseq);

      if (after(ntohl(tcph->ack_seq) - other_way->offset_before,
              other_way->correction_pos))
            newack = ntohl(tcph->ack_seq) - other_way->offset_after;
      else
            newack = ntohl(tcph->ack_seq) - other_way->offset_before;
      newack = htonl(newack);

      tcph->check = ip6_int_nat_cheat_check(~tcph->seq, newseq,
                               ip6_int_nat_cheat_check(~tcph->ack_seq,
                                              newack,
                                              tcph->check));

      DEBUGP("Adjusting sequence number from %u->%u, ack from %u-
>%u\n",
            ntohl(tcph->seq), ntohl(newseq), ntohl(tcph->ack_seq),
            ntohl(newack));

      tcph->seq = newseq;
      tcph->ack_seq = newack;

      return ip6_nat_sack_adjust(pskb, tcph, ct, ctinfo);
}

static inline int
helper_cmp(const struct ip6_nat_helper *helper,
         const struct ip6_conntrack_tuple *tuple)
{
      return   ip6_ct_tuple_mask_cmp(tuple,   &helper->tuple,   &helper-
>mask);
}

int ip6_nat_helper_register(struct ip6_nat_helper *me)
{
      int ret = 0;

      WRITE_LOCK(&ip6_nat_lock);
      if  (LIST_FIND(&ip6_helpers,  helper_cmp,  struct  ip6_nat_helper
*,&me->tuple))
            ret = -EBUSY;
      else
            list_prepend(&ip6_helpers, me);
      WRITE_UNLOCK(&ip6_nat_lock);

      return ret;
}

static int
kill_helper(const struct ip6_conntrack *i, void *helper)
{
      int ret;

      READ_LOCK(&ip6_nat_lock);
      ret = (i->nat.info.helper == helper);
      READ_UNLOCK(&ip6_nat_lock);

      return ret;
}

void ip6_nat_helper_unregister(struct ip6_nat_helper *me)
{
      WRITE_LOCK(&ip6_nat_lock);
      /* Autoloading conntrack helper might have failed */
      if  (LIST_FIND(&ip6_helpers,  helper_cmp,  struct  ip6_nat_helper
*,&me->tuple)) {
            LIST_DELETE(&ip6_helpers, me);
      }
      WRITE_UNLOCK(&ip6_nat_lock);

      /* Someone could be still looking at the helper in a bh. */
      synchronize_net();

      /* Find anything using it, and umm, kill them.  We can't turn
         them into normal connections: if we've adjusted SYNs, then
         they'll ackstorm.  So we just drop it.  We used to just
         bump module count when a connection existed, but that
         forces admins to gen fake RSTs or bounce box, either of
         which is just a long-winded way of making things
         worse. --RR */
      ip6_ct_selective_cleanup(kill_helper, me);
}
```

/INCLUDE/LINUX/NETFILTER_IPV6/IP6_NAT_HELPER.H

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: include/linux/ip_nat_helper.h
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * This file was ported, yet due to the scope of the thesis, no helper
 * files were used. This file was not tested.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

#ifndef _IP6_NAT_HELPER_H
#define _IP6_NAT_HELPER_H
/* NAT protocol helper routines. */

#include <linux/netfilter_ipv6/ip6_conntrack.h>
#include <linux/module.h>

struct sk_buff;

/* Flags */
/* NAT helper must be called on every packet (for TCP) */
#define IP6_NAT_HELPER_F_ALWAYS           0x01

struct ip6_nat_helper
{
      struct list_head list;        /* Internal use */

      const char *name;       /* name of the module */
      unsigned char flags;          /* Flags (see above) */
      struct module *me;            /* pointer to self */

      /* Mask of things we will help: vs. tuple from server */
      struct ip6_conntrack_tuple tuple;
      struct ip6_conntrack_tuple mask;
      /* Helper function: returns verdict */
      unsigned int (*help)(struct ip6_conntrack *ct,
                       struct ip6_conntrack_expect *exp,
                       struct ip6_nat_info *info,
                       enum ip6_conntrack_info ctinfo,
                       unsigned int hooknum,
                       struct sk_buff **pskb);

      /* Returns verdict and sets up NAT for this connection */
      unsigned int (*expect)(struct sk_buff **pskb,
                         unsigned int hooknum,
                         struct ip6_conntrack *ct,
                         struct ip6_nat_info *info);
};

extern struct list_head ip6_helpers;

extern int ip6_nat_helper_register(struct ip6_nat_helper *me);
extern void ip6_nat_helper_unregister(struct ip6_nat_helper *me);

/* These return true or false. */
extern int ip6_nat_mangle_tcp_packet(struct sk_buff **skb,
                        struct ip6_conntrack *ct,
                        enum ip6_conntrack_info ctinfo,
                        unsigned int match_offset,
                        unsigned int match_len,
                        const char *rep_buffer,
                        unsigned int rep_len);
extern int ip6_nat_mangle_udp_packet(struct sk_buff **skb,
                        struct ip6_conntrack *ct,
                        enum ip6_conntrack_info ctinfo,
                        unsigned int match_offset,
                        unsigned int match_len,
                        const char *rep_buffer,
                        unsigned int rep_len);
extern int ip6_nat_seq_adjust(struct sk_buff **pskb,
                       struct ip6_conntrack *ct,
                       enum ip6_conntrack_info ctinfo);
#endif
```

/NET/IPV6/NETFILTER/IP6_NAT_PROTO_ICMP.C

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: net/ipv4/netfilter/ip_nat_proto_icmp.c
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

/* (C) 1999-2001 Paul `Rusty' Russell
 * (C) 2002-2004 Netfilter Core Team <coreteam@netfilter.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */

#include <linux/types.h>
#include <linux/init.h>
#include <linux/netfilter.h>
#include <linux/ipv6.h>
#include <linux/icmpv6.h>
#include <linux/if.h>
#include <net/checksum.h>

#include <linux/netfilter_ipv6/ip6_nat.h>
#include <linux/netfilter_ipv6/ip6_nat_core.h>
#include <linux/netfilter_ipv6/ip6_nat_rule.h>
#include <linux/netfilter_ipv6/ip6_nat_protocol.h>

static int
icmpv6_in_range(const struct ip6_conntrack_tuple *tuple,
            enum ip6_nat_manip_type maniptype,
            const union ip6_conntrack_manip_proto *min,
            const union ip6_conntrack_manip_proto *max)
{
      return (tuple->src.u.icmpv6.id >= min->icmpv6.id
            && tuple->src.u.icmpv6.id <= max->icmpv6.id);
}

static int
icmpv6_unique_tuple(struct ip6_conntrack_tuple *tuple,
              const struct ip6_nat_range *range,
              enum ip6_nat_manip_type maniptype,
              const struct ip6_conntrack *conntrack)
{
      static u_int16_t id;
      unsigned int range_size
            = (unsigned int)range->max.icmpv6.id - range->min.icmpv6.id
+ 1;
      unsigned int i;

      /* If no range specified... */
      if (!(range->flags & IP6_NAT_RANGE_PROTO_SPECIFIED))
            range_size = 0xFFFF;

      for (i = 0; i < range_size; i++, id++) {
            tuple->src.u.icmpv6.id  =  range->min.icmpv6.id  +  (id  %
range_size);
            if (!ip6_nat_used_tuple(tuple, conntrack))
                  return 1;
      }
      return 0;
}

static int
icmpv6_manip_pkt(struct sk_buff **pskb,
             unsigned int hdroff,
             const struct ip6_conntrack_manip *manip,
             enum ip6_nat_manip_type maniptype)
{

struct sk_buff *skb = *pskb;
      struct icmp6hdr *hdr;


 if (!skb_ip6_make_writable(pskb, hdroff + sizeof(hdr))){

            return 0;
 }
      hdr = (void *)(*pskb)->data + hdroff;

      struct in6_addr *saddr, *daddr;

      saddr = &skb->nh.ipv6h->saddr;
        daddr = &skb->nh.ipv6h->daddr;

      hdr->icmp6_cksum = 0;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - Here we use the csum_ipv6_magic and csum_partial functions
to calculate the
```

```
 * ICMPv6 header checksum.  csum_partial determines the checksum for
just the ICMPv6 header
 * but does not flip the bits at the end.  This is then folded into the
pseudo-header checksum
 * calculation done by csum_ipv6_magic, which then yields a proper
checksum for the entire
 * ICMPv6 header and pseudo-header combination.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */
      hdr->icmp6_cksum = csum_ipv6_magic(saddr,
                                 daddr,   (*pskb)->len - sizeof(struct
ipv6hdr),
                                 IPPROTO_ICMPV6,
csum_partial((char *)hdr, (*pskb)->len - sizeof(struct ipv6hdr), 0));


      hdr->icmp6_dataun.u_echo.identifier = manip->u.icmpv6.id;

      return 1;
}

static unsigned int
icmpv6_print(char *buffer,
         const struct ip6_conntrack_tuple *match,
         const struct ip6_conntrack_tuple *mask)
{
      unsigned int len = 0;

      if (mask->src.u.icmpv6.id)
            len += sprintf(buffer + len, "id=%u ",
                         ntohs(match->src.u.icmpv6.id));

      if (mask->dst.u.icmpv6.type)
            len += sprintf(buffer + len, "type=%u ",
                         ntohs(match->dst.u.icmpv6.type));

      if (mask->dst.u.icmpv6.code)
            len += sprintf(buffer + len, "code=%u ",
                         ntohs(match->dst.u.icmpv6.code));

      return len;
}

static unsigned int
icmpv6_print_range(char *buffer, const struct ip6_nat_range *range)
{
      if (range->min.icmpv6.id != 0 || range->max.icmpv6.id != 0xFFFF)
            return sprintf(buffer, "id %u-%u ",
                         ntohs(range->min.icmpv6.id),
                         ntohs(range->max.icmpv6.id));
      else return 0;
}

struct ip6_nat_protocol ip6_nat_protocol_icmp
= { { NULL, NULL }, "ICMP", IPPROTO_ICMPV6,
    icmpv6_manip_pkt,
    icmpv6_in_range,
    icmpv6_unique_tuple,
    icmpv6_print,
    icmpv6_print_range,
};
```

/NET/IPV6/NETFILTER/IP6_NAT_PROTO_TCP.C

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: net/ipv4/netfilter/ip_nat_proto_tcp.c
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

/* (C) 1999-2001 Paul `Rusty' Russell
 * (C) 2002-2004 Netfilter Core Team <coreteam@netfilter.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */

#include <linux/types.h>
#include <linux/init.h>
#include <linux/netfilter.h>
#include <linux/ipv6.h>
#include <linux/tcp.h>
#include <linux/if.h>
#include <net/checksum.h>
#include <linux/netfilter_ipv6/ip6_nat.h>
#include <linux/netfilter_ipv6/ip6_nat_rule.h>
#include <linux/netfilter_ipv6/ip6_nat_protocol.h>
#include <linux/netfilter_ipv6/ip6_nat_core.h>

static int
tcp_in_range(const struct ip6_conntrack_tuple *tuple,
           enum ip6_nat_manip_type maniptype,
           const union ip6_conntrack_manip_proto *min,
           const union ip6_conntrack_manip_proto *max)
{
      u_int16_t port;
      if (maniptype == IP6_NAT_MANIP_SRC)
            port = tuple->src.u.tcp.port;
      else
            port = tuple->dst.u.tcp.port;

      return ntohs(port) >= ntohs(min->tcp.port)
            && ntohs(port) <= ntohs(max->tcp.port);
}

static int
tcp_unique_tuple(struct ip6_conntrack_tuple *tuple,
             const struct ip6_nat_range *range,
             enum ip6_nat_manip_type maniptype,
             const struct ip6_conntrack *conntrack)
{
      static u_int16_t port, *portptr;
      unsigned int range_size, min, i;

      if (maniptype == IP6_NAT_MANIP_SRC)
            portptr = &tuple->src.u.tcp.port;
      else
            portptr = &tuple->dst.u.tcp.port;

      /* If no range specified... */
      if (!(range->flags & IP6_NAT_RANGE_PROTO_SPECIFIED)) {
            /* If it's dst rewrite, can't change port */
            if (maniptype == IP6_NAT_MANIP_DST)
                  return 0;

            /* Map privileged onto privileged. */
            if (ntohs(*portptr) < 1024) {
                  /* Loose convention: >> 512 is credential passing */
                  if (ntohs(*portptr)<512) {
                        min = 1;
                        range_size = 511 - min + 1;
                  } else {
                        min = 600;
                        range_size = 1023 - min + 1;
                  }
            } else {
                  min = 1024;
                  range_size = 65535 - 1024 + 1;
            }
      } else {
            min = ntohs(range->min.tcp.port);
            range_size = ntohs(range->max.tcp.port) - min + 1;
      }

      for (i = 0; i < range_size; i++, port++) {
            *portptr = htons(min + port % range_size);
            if (!ip6_nat_used_tuple(tuple, conntrack)) {
                  return 1;
            }
      }
      return 0;
}
```

```
static int
tcp_manip_pkt(struct sk_buff **pskb,
            unsigned int hdroff,
            const struct ip6_conntrack_manip *manip,
            enum ip6_nat_manip_type maniptype)
{
      struct tcphdr *hdr;
      struct in6_addr oldip;
      u_int16_t *portptr, oldport;
      int hdrsize = 8; /* TCP connection tracking guarantees this much
*/

      /* this could be a inner header returned in icmp packet; in such
         cases we cannot update the checksum field since it is outside
of
         the 8 bytes of transport layer headers we are guaranteed */
      if ((*pskb)->len >= hdroff + sizeof(struct tcphdr))
            hdrsize = sizeof(struct tcphdr);

      if (!skb_ip6_make_writable(pskb, hdroff + hdrsize))
            return 0;

      hdr = (void *)(*pskb)->data + hdroff;

      if (maniptype == IP6_NAT_MANIP_SRC) {
            /* Get rid of src ip and src pt */
            oldip = (*pskb)->nh.ipv6h->saddr;

            portptr = &hdr->source;
      } else {
            /* Get rid of dst ip and dst pt */
            oldip = (*pskb)->nh.ipv6h->daddr;
            portptr = &hdr->dest;
      }

      oldport = *portptr;
      *portptr = manip->u.tcp.port;

      if (hdrsize < sizeof(*hdr))
        return 1;

      hdr->check = 0;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - Here we use the csum_ipv6_magic and csum_partial functions
to calculate the
 * TCP header checksum.  csum_partial determines the checksum for just
the TCP header
 * but does not flip the bits at the end.  This is then folded into the
pseudo-header checksum
 * calculation done by csum_ipv6_magic, which then yields a proper
checksum for the entire
 * TCP header and pseudo-header combination.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */

 hdr->check = csum_ipv6_magic(&(*pskb)->nh.ipv6h->saddr,
                              &(*pskb)->nh.ipv6h->daddr,
                              (*pskb)->len - sizeof(struct ipv6hdr),
                              IPPROTO_TCP,
                              csum_partial((char *)hdr, (*pskb)->len -
sizeof(struct ipv6hdr), 0));

return 1;
}

static unsigned int
tcp_print(char *buffer,
    const struct ip6_conntrack_tuple *match,
    const struct ip6_conntrack_tuple *mask)
{
   unsigned int len = 0;

   if (mask->src.u.tcp.port)
      len += sprintf(buffer + len, "srcpt=%u ",
            ntohs(match->src.u.tcp.port));


   if (mask->dst.u.tcp.port)
      len += sprintf(buffer + len, "dstpt=%u ",
            ntohs(match->dst.u.tcp.port));

   return len;
}

static unsigned int
tcp_print_range(char *buffer, const struct ip6_nat_range *range)
{
   if (range->min.tcp.port != 0 || range->max.tcp.port != 0xFFFF) {
      if (range->min.tcp.port == range->max.tcp.port)
         return sprintf(buffer, "port %u ",
               ntohs(range->min.tcp.port));
      else
         return sprintf(buffer, "ports %u-%u ",
               ntohs(range->min.tcp.port),
               ntohs(range->max.tcp.port));
   }
   else return 0;
}

struct ip6_nat_protocol ip6_nat_protocol_tcp
= { { NULL, NULL }, "TCP", IPPROTO_TCP,
  tcp_manip_pkt,
  tcp_in_range,
  tcp_unique_tuple,
  tcp_print,
  tcp_print_range
};
```

/NET/IPV6/NETFILTER/IP6_NAT_PROTO_UDP.C

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: net/ipv4/netfilter/ip_nat_proto_udp.c
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

/* (C) 1999-2001 Paul `Rusty' Russell
 * (C) 2002-2004 Netfilter Core Team <coreteam@netfilter.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */

#include <linux/types.h>
#include <linux/init.h>
#include <linux/netfilter.h>
#include <linux/ipv6.h>
#include <linux/udp.h>
#include <linux/if.h>
#include <net/checksum.h>

#include <linux/netfilter_ipv6/ip6_nat.h>
#include <linux/netfilter_ipv6/ip6_nat_core.h>
#include <linux/netfilter_ipv6/ip6_nat_rule.h>
#include <linux/netfilter_ipv6/ip6_nat_protocol.h>

static int
udp_in_range(const struct ip6_conntrack_tuple *tuple,
           enum ip6_nat_manip_type maniptype,
           const union ip6_conntrack_manip_proto *min,
           const union ip6_conntrack_manip_proto *max)
{
        u_int16_t port;
        if (maniptype == IP6_NAT_MANIP_SRC)
              port = tuple->src.u.udp.port;
        else
              port = tuple->dst.u.udp.port;

        return ntohs(port) >= ntohs(min->udp.port)
              && ntohs(port) <= ntohs(max->udp.port);
}

static int
udp_unique_tuple(struct ip6_conntrack_tuple *tuple,
             const struct ip6_nat_range *range,
             enum ip6_nat_manip_type maniptype,
             const struct ip6_conntrack *conntrack)
{

        static u_int16_t port, *portptr;
        unsigned int range_size, min, i;

        if (maniptype == IP6_NAT_MANIP_SRC)
              portptr = &tuple->src.u.udp.port;
        else
              portptr = &tuple->dst.u.udp.port;

        /* If no range specified... */
        if (!(range->flags & IP6_NAT_RANGE_PROTO_SPECIFIED)) {
              /* If it's dst rewrite, can't change port */
              if (maniptype == IP6_NAT_MANIP_DST)
                    return 0;

              if (ntohs(*portptr) < 1024) {
                    /* Loose convention: >> 512 is credential passing */
                    if (ntohs(*portptr)<512) {
                          min = 1;
                          range_size = 511 - min + 1;
                    } else {
                          min = 600;
                          range_size = 1023 - min + 1;
                    }
              } else {
                    min = 1024;
                    range_size = 65535 - 1024 + 1;
              }
        } else {
              min = ntohs(range->min.udp.port);
              range_size = ntohs(range->max.udp.port) - min + 1;
        }

        for (i = 0; i < range_size; i++, port++) {
              *portptr = htons(min + port % range_size);
              if (!ip6_nat_used_tuple(tuple, conntrack))
                    return 1;
        }
        return 0;
}
```

```
static int
udp_manip_pkt(struct sk_buff **pskb,
            unsigned int hdroff,
            const struct ip6_conntrack_manip *manip,
            enum ip6_nat_manip_type maniptype)
{

      struct udphdr *hdr;
      struct in6_addr oldip;
      u_int16_t *portptr;

      if (!skb_ip6_make_writable(pskb, hdroff + sizeof(hdr)))
            return 0;

      hdr = (void *)(*pskb)->data + hdroff;
      if (maniptype == IP6_NAT_MANIP_SRC) {
            /* Get rid of src ip and src pt */
            oldip = (*pskb)->nh.ipv6h->saddr;
            portptr = &hdr->source;
      } else {
            /* Get rid of dst ip and dst pt */
            oldip = (*pskb)->nh.ipv6h->daddr;
            portptr = &hdr->dest;
      }
      if (hdr->check){ /* 0 is a special case meaning no checksum */

        hdr->check = 0;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - Here we use the csum_ipv6_magic and csum_partial functions
to calculate the
 * UDP header checksum.  csum_partial determines the checksum for just
the UDP header
 * but does not flip the bits at the end.  This is then folded into the
pseudo-header checksum
 *  calculation  done  by  csum_ipv6_magic,  which  then  yields  a  proper
checksum for the entire
 * UDP header and pseudo-header combination.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * */

        hdr->check = csum_ipv6_magic(&(*pskb)->nh.ipv6h->saddr,
                               &(*pskb)->nh.ipv6h->daddr,
                               (*pskb)->len - sizeof(struct ipv6hdr),
                               IPPROTO_UDP,
                               csum_partial((char *)hdr, (*pskb)->len -
sizeof(struct ipv6hdr), 0));

      }

      *portptr = manip->u.udp.port;
      return 1;
}
```

```
static unsigned int
udp_print(char *buffer,
        const struct ip6_conntrack_tuple *match,
        const struct ip6_conntrack_tuple *mask)
{

      unsigned int len = 0;

      if (mask->src.u.udp.port)
            len += sprintf(buffer + len, "srcpt=%u ",
                         ntohs(match->src.u.udp.port));


      if (mask->dst.u.udp.port)
            len += sprintf(buffer + len, "dstpt=%u ",
                         ntohs(match->dst.u.udp.port));

      return len;
}

static unsigned int
udp_print_range(char *buffer, const struct ip6_nat_range *range)
{

      if (range->min.udp.port != 0 || range->max.udp.port != 0xFFFF) {
            if (range->min.udp.port == range->max.udp.port)
                  return sprintf(buffer, "port %u ",
                               ntohs(range->min.udp.port));
            else
                  return sprintf(buffer, "ports %u-%u ",
                               ntohs(range->min.udp.port),
                               ntohs(range->max.udp.port));
      }
      else return 0;
}

struct ip6_nat_protocol ip6_nat_protocol_udp
= { { NULL, NULL }, "UDP", IPPROTO_UDP,
    udp_manip_pkt,
    udp_in_range,
    udp_unique_tuple,
    udp_print,
    udp_print_range
};
```

/NET/IPV6/NETFILTER/IP6_NAT_PROTO_UNKNOWN.C

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: net/ipv4/netfilter/ip_nat_proto_unknown.c
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

/* The "unknown" protocol.  This is what is used for protocols we
 * don't understand.  It's returned by ip_ct_find_proto().
 */

/* (C) 1999-2001 Paul `Rusty' Russell
 * (C) 2002-2004 Netfilter Core Team <coreteam@netfilter.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */

#include <linux/types.h>
#include <linux/init.h>
#include <linux/netfilter.h>
#include <linux/if.h>

#include <linux/netfilter_ipv6/ip6_nat.h>
#include <linux/netfilter_ipv6/ip6_nat_rule.h>
#include <linux/netfilter_ipv6/ip6_nat_protocol.h>

static int unknown_in_range(const struct ip6_conntrack_tuple *tuple,
                      enum ip6_nat_manip_type manip_type,
                      const union ip6_conntrack_manip_proto *min,
                      const union ip6_conntrack_manip_proto *max)
{
      return 1;
}

static int unknown_unique_tuple(struct ip6_conntrack_tuple *tuple,
                        const struct ip6_nat_range *range,
                        enum ip6_nat_manip_type maniptype,
                        const struct ip6_conntrack *conntrack)
{
      /* Sorry: we can't help you; if it's not unique, we can't frob
         anything. */
      return 0;
}

static int
unknown_manip_pkt(struct sk_buff **pskb,
              unsigned int hdroff,
              const struct ip6_conntrack_manip *manip,
              enum ip6_nat_manip_type maniptype)
{
      return 1;
}

static unsigned int
unknown_print(char *buffer,
            const struct ip6_conntrack_tuple *match,
            const struct ip6_conntrack_tuple *mask)
{
      return 0;
}

static unsigned int
unknown_print_range(char *buffer, const struct ip6_nat_range *range)
{
      return 0;
}

struct ip6_nat_protocol ip6_unknown_nat_protocol = {
      { NULL, NULL }, "unknown", 0,
      unknown_manip_pkt,
      unknown_in_range,
      unknown_unique_tuple,
      unknown_print,
      unknown_print_range
};
```

/INCLUDE/LINUX/NETFILTER_IPV6/IP6_NAT_PROTOCOL.H

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: include/linux/ip_nat_protocol.h
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */


/* Header for use in defining a given protocol. */
#ifndef _IP6_NAT_PROTOCOL_H
#define _IP6_NAT_PROTOCOL_H
#include <linux/init.h>
#include <linux/list.h>

struct ipv6hdr;
struct ip6_nat_range;

struct ip6_nat_protocol
{
      struct list_head list;

      /* Protocol name */
      const char *name;

      /* Protocol number. */
      unsigned int protonum;

      /* Do a packet translation according to the ip_nat_proto_manip
       * and manip type.  Return true if succeeded. */
      int (*manip_pkt)(struct sk_buff **pskb,
                   unsigned int hdroff,
                   const struct ip6_conntrack_manip *manip,
                   enum ip6_nat_manip_type maniptype);
      /* Is the manipable part of the tuple between min and max incl?
*/
      int (*in_range)(const struct ip6_conntrack_tuple *tuple,
                  enum ip6_nat_manip_type maniptype,
                  const union ip6_conntrack_manip_proto *min,
                  const union ip6_conntrack_manip_proto *max);

      /* Alter the per-proto part of the tuple (depending on
         maniptype), to give a unique tuple in the given range if
         possible; return false if not.  Per-protocol part of tuple
         is initialized to the incoming packet. */
      int (*unique_tuple)(struct ip6_conntrack_tuple *tuple,
                      const struct ip6_nat_range *range,
                      enum ip6_nat_manip_type maniptype,
                      const struct ip6_conntrack *conntrack);

      unsigned int (*print)(char *buffer,
                        const struct ip6_conntrack_tuple *match,
                        const struct ip6_conntrack_tuple *mask);

      unsigned int (*print_range)(char *buffer,
                            const struct ip6_nat_range *range);
};

/* Protocol registration. */
extern int ip6_nat_protocol_register(struct ip6_nat_protocol *proto);
extern    void    ip6_nat_protocol_unregister(struct    ip6_nat_protocol
*proto);

extern int init_protocols(void) __init;
extern void cleanup_protocols(void);
extern struct ip6_nat_protocol *ip6_find_nat_proto(u_int16_t protonum);

#endif /*_IP6_NAT_PROTO_H*/
```

/NET/IPV6/NETFILTER/IP6_NAT_RULE.C

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: net/ipv4/netfilter/ip_nat_rule.c
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

/* (C) 1999-2001 Paul `Rusty' Russell
 * (C) 2002-2004 Netfilter Core Team <coreteam@netfilter.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */

/* Everything about the rules for NAT. */
#include <linux/types.h>
#include <linux/ipv6.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv6.h>
#include <linux/module.h>
#include <linux/kmod.h>
#include <linux/skbuff.h>
#include <linux/proc_fs.h>
#include <net/checksum.h>
#include <linux/bitops.h>

#define ASSERT_READ_LOCK(x) MUST_BE_READ_LOCKED(&ip6_nat_lock)
#define ASSERT_WRITE_LOCK(x) MUST_BE_WRITE_LOCKED(&ip6_nat_lock)

#if 0
#define DEBUGP printk
#else
#define DEBUGP(format, args...)
```

```
#endif

#include <linux/netfilter_ipv6/ip6_tables.h>
#include <linux/netfilter_ipv6/ip6_nat.h>
#include <linux/netfilter_ipv6/ip6_nat_core.h>
#include <linux/netfilter_ipv6/ip6_nat_rule.h>
#include <linux/netfilter_ipv4/listhelp.h>

#define         NAT_VALID_HOOKS         ((1<<NF_IP6_PRE_ROUTING)         |
(1<<NF_IP6_POST_ROUTING) | (1<<NF_IP6_LOCAL_OUT))

/* Standard entry. */
struct ip6t_standard
{
      struct ip6t_entry entry;
      struct ip6t_standard_target target;
};

struct ip6t_error_target
{
      struct ip6t_entry_target target;
      char errorname[IP6T_FUNCTION_MAXNAMELEN];
};

struct ip6t_error
{
      struct ip6t_entry entry;
      struct ip6t_error_target target;
};

static struct
{
      struct ip6t_replace repl;
      struct ip6t_standard entries[3];
      struct ip6t_error term;
} nat_initial_table __initdata
= { { "nat", NAT_VALID_HOOKS, 4,
      sizeof(struct ip6t_standard) * 3 + sizeof(struct ip6t_error),
      { [NF_IP6_PRE_ROUTING] = 0,
      [NF_IP6_POST_ROUTING] = sizeof(struct ip6t_standard),
      [NF_IP6_LOCAL_OUT] = sizeof(struct ip6t_standard) * 2 },
      { [NF_IP6_PRE_ROUTING] = 0,
      [NF_IP6_POST_ROUTING] = sizeof(struct ip6t_standard),
      [NF_IP6_LOCAL_OUT] = sizeof(struct ip6t_standard) * 2 },
      0, NULL, { } },
    {
          /* PRE_ROUTING */
          { { { { { { 0 } } }, { { { 0 } } }, { { { 0 } } }, { { { 0 }
} }, "", "", { 0 }, { 0 }, 0, 0, 0 },
            0,
            sizeof(struct ip6t_entry),
            sizeof(struct ip6t_standard),
            0, { 0, 0 }, { } },
            { { { { IP6T_ALIGN(sizeof(struct ip6t_standard_target)), ""
} }, { } },
            -NF_ACCEPT - 1 } },
          /* POST_ROUTING */
          { { { { { { 0 } } }, { { { 0 } } }, { { { 0 } } }, { { { 0 }
} }, "", "", { 0 }, { 0 }, 0, 0, 0 },
            0,
            sizeof(struct ip6t_entry),
            sizeof(struct ip6t_standard),
            0, { 0, 0 }, { } },
            { { { { IP6T_ALIGN(sizeof(struct ip6t_standard_target)), ""
} }, { } },
            -NF_ACCEPT - 1 } },
          /* LOCAL_OUT */
          { { { { { { 0 } } }, { { { 0 } } }, { { { 0 } } }, { { { 0 }
} }, "", "", { 0 }, { 0 }, 0, 0, 0 },
            0,
            sizeof(struct ip6t_entry),
            sizeof(struct ip6t_standard),
            0, { 0, 0 }, { } },
            { { { { IP6T_ALIGN(sizeof(struct ip6t_standard_target)), ""
} }, { } },
            -NF_ACCEPT - 1 } }
    },
    /* ERROR */
    { { { { { { 0 } } }, { { { 0 } } }, { { { 0 } } }, { { { 0 } } },
"", "", { 0 }, { 0 }, 0, 0, 0 },
      0,
      sizeof(struct ip6t_entry),
      sizeof(struct ip6t_error),
      0, { 0, 0 }, { } },
      {   {   {   {   IP6T_ALIGN(sizeof(struct   ip6t_error_target)),
IP6T_ERROR_TARGET } },
        { } },
      "ERROR"
      }
    }
};

static struct ip6t_table nat_table = {
      .name       = "nat",
      .table            = &nat_initial_table.repl,
      .valid_hooks      = NAT_VALID_HOOKS,
      .lock       = RW_LOCK_UNLOCKED,
      .me         = THIS_MODULE,
};

/* Source NAT */
static unsigned int ip6t_snat_target(struct sk_buff **pskb,
                             unsigned int hooknum,
                            const struct net_device *in,
                            const struct net_device *out,
                            const void *targinfo,
                            void *userinfo)
{

      struct ip6_conntrack *ct;
      enum ip6_conntrack_info ctinfo;

      IP6_NF_ASSERT(hooknum == NF_IP6_POST_ROUTING);
      ct = ip6_conntrack_get(*pskb, &ctinfo);

      /* Connection must be valid and new. */
      IP6_NF_ASSERT(ct   &&   (ctinfo   ==   IP6_CT_NEW   ||   ctinfo   ==
IP6_CT_RELATED));

      IP6_NF_ASSERT(out);

return ip6_nat_setup_info(ct, targinfo, hooknum);
}

static unsigned int ip6t_dnat_target(struct sk_buff **pskb,
                            unsigned int hooknum,
                            const struct net_device *in,
                            const struct net_device *out,
                            const void *targinfo,
                            void *userinfo)
{
      struct ip6_conntrack *ct;
      enum ip6_conntrack_info ctinfo;

#ifdef CONFIG_IP6_NF_NAT_LOCAL
      IP6_NF_ASSERT(hooknum == NF_IP6_PRE_ROUTING
                 || hooknum == NF_IP6_LOCAL_OUT);
#else
      IP6_NF_ASSERT(hooknum == NF_IP6_PRE_ROUTING);
#endif

      ct = ip6_conntrack_get(*pskb, &ctinfo);

      /* Connection must be valid and new. */
      IP6_NF_ASSERT(ct   &&   (ctinfo   ==   IP6_CT_NEW   ||   ctinfo   ==
IP6_CT_RELATED));

      return ip6_nat_setup_info(ct, targinfo, hooknum);
}

static int ip6t_snat_checkentry(const char *tablename,
                         const struct ip6t_entry *e,
                         void *targinfo,
                         unsigned int targinfosize,
                         unsigned int hook_mask)
{
      struct ip6_nat_multi_range *mr = targinfo;

      /* Must be a valid range */
      if (targinfosize < sizeof(struct ip6_nat_multi_range)) {
            DEBUGP("SNAT: Target size %u too small\n", targinfosize);
            return 0;
      }

      if        (targinfosize        !=        IP6T_ALIGN((sizeof(struct
ip6_nat_multi_range)
                               + (sizeof(struct ip6_nat_range)
                                * (mr->rangesize - 1))))) {
            DEBUGP("SNAT: Target size %u wrong for %u ranges\n",
                   targinfosize, mr->rangesize);
```

```
		return 0;
	}

	/* Only allow these for NAT. */
	if (strcmp(tablename, "nat") != 0) {
		DEBUGP("SNAT: wrong table %s\n", tablename);
		return 0;
	}

	if (hook_mask & ~(1 << NF_IP6_POST_ROUTING)) {
		DEBUGP("SNAT: hook mask 0x%x bad\n", hook_mask);
		return 0;
	}
	return 1;
}

static int ip6t_dnat_checkentry(const char *tablename,
			 const struct ip6t_entry *e,
			 void *targinfo,
			 unsigned int targinfosize,
			 unsigned int hook_mask)
{
	struct ip6_nat_multi_range *mr = targinfo;

	/* Must be a valid range */
	if (targinfosize < sizeof(struct ip6_nat_multi_range)) {
		DEBUGP("DNAT: Target size %u too small\n", targinfosize);
		return 0;
	}

	if        (targinfosize        !=        IP6T_ALIGN((sizeof(struct
ip6_nat_multi_range)
				+ (sizeof(struct ip6_nat_range)
				 * (mr->rangesize - 1))))) {
		DEBUGP("DNAT: Target size %u wrong for %u ranges\n",
		       targinfosize, mr->rangesize);
		return 0;
	}

	/* Only allow these for NAT. */
	if (strcmp(tablename, "nat") != 0) {
		DEBUGP("DNAT: wrong table %s\n", tablename);
		return 0;
	}

	if   (hook_mask   &   ~((1   <<   NF_IP6_PRE_ROUTING)   |   (1   <<
NF_IP6_LOCAL_OUT))) {
		DEBUGP("DNAT: hook mask 0x%x bad\n", hook_mask);
		return 0;
	}

#ifndef CONFIG_IP6_NF_NAT_LOCAL
	if (hook_mask & (1 << NF_IP6_LOCAL_OUT)) {
		DEBUGP("DNAT: CONFIG_IP6_NF_NAT_LOCAL not enabled\n");
		return 0;
	}
#endif
```

```
      return 1;
}

inline unsigned int
ip6_alloc_null_binding(struct ip6_conntrack *conntrack,
               struct ip6_nat_info *info,
               unsigned int hooknum)
{

      /* Force range to this IP; let proto decide mapping for
         per-proto parts (hence not IP_NAT_RANGE_PROTO_SPECIFIED).
         Use reply in case it's already been mangled (eg local packet).
      */

      struct in6_addr ip
            = (HOOK2MANIP(hooknum) == IP6_NAT_MANIP_SRC
               ? conntrack->tuplehash[IP6_CT_DIR_REPLY].tuple.dst.ip
               : conntrack->tuplehash[IP6_CT_DIR_REPLY].tuple.src.ip);
      struct ip6_nat_multi_range mr
            = { 1, { { IP6_NAT_RANGE_MAP_IPS, ip, ip, { 0 }, { 0 } } }
};

      DEBUGP("Allocating      NULL      binding      for      %p      (
%x:%x:%x:%x:%x:%x:%x:%x)\n", conntrack,
             NIP6(ip));

      return ip6_nat_setup_info(conntrack, &mr, hooknum);
}

int ip6_nat_rule_find(struct sk_buff **pskb,
                 unsigned int hooknum,
                 const struct net_device *in,
                 const struct net_device *out,
                 struct ip6_conntrack *ct,
                 struct ip6_nat_info *info)
{
  int ret;

      ret = ip6t_do_table(pskb, hooknum, in, out, &nat_table, NULL);

      if (ret == NF_ACCEPT) {

        if (!(info->initialized & (1 << HOOK2MANIP(hooknum)))){

      /* NUL mapping */
                  ret = ip6_alloc_null_binding(ct, info, hooknum);
        }
      }

      return ret;
}

static struct ip6t_target ip6t_snat_reg = {
      .name       = "SNAT",
      .target           = ip6t_snat_target,
      .checkentry = ip6t_snat_checkentry,
};

static struct ip6t_target ip6t_dnat_reg = {
      .name       = "DNAT",
      .target           = ip6t_dnat_target,
      .checkentry = ip6t_dnat_checkentry,
};

int __init ip6_nat_rule_init(void)
{
      int ret;

      ret = ip6t_register_table(&nat_table);
      if (ret != 0)
            return ret;
      ret = ip6t_register_target(&ip6t_snat_reg);
      if (ret != 0)
            goto unregister_table;

      ret = ip6t_register_target(&ip6t_dnat_reg);
      if (ret != 0)
            goto unregister_snat;

      return ret;

 unregister_snat:
      ip6t_unregister_target(&ip6t_snat_reg);
 unregister_table:
      ip6t_unregister_table(&nat_table);

      return ret;
}

void ip6_nat_rule_cleanup(void)
{
      ip6t_unregister_target(&ip6t_dnat_reg);
      ip6t_unregister_target(&ip6t_snat_reg);
      ip6t_unregister_table(&nat_table);
}
```

/INCLUDE/LINUX/NETFILTER_IPV6/IP6_NAT_RULE.H

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: include/linux/ip_nat_rule.h
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */


#ifndef _IP6_NAT_RULE_H
#define _IP6_NAT_RULE_H
#include <linux/netfilter_ipv6/ip6_conntrack.h>
#include <linux/netfilter_ipv6/ip6_tables.h>
#include <linux/netfilter_ipv6/ip6_nat.h>

#ifdef __KERNEL__

extern int ip6_nat_rule_init(void) __init;
extern void ip6_nat_rule_cleanup(void);
extern int ip6_nat_rule_find(struct sk_buff **pskb,
                      unsigned int hooknum,
                      const struct net_device *in,
                      const struct net_device *out,
                      struct ip6_conntrack *ct,
                      struct ip6_nat_info *info);

extern unsigned int
ip6_alloc_null_binding(struct ip6_conntrack *conntrack,
               struct ip6_nat_info *info,
               unsigned int hooknum);
#endif
#endif /* _IP6_NAT_RULE_H */
```

/NET/IPV6/NETFILTER/IP6_NAT_STANDALONE.C

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: net/ipv4/netfilter/ip_nat_standalone.c
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

/* This file contains all the functions required for the standalone
   ip_nat module.

   These are not required by the compatibility layer.
*/

/* (C) 1999-2001 Paul `Rusty' Russell
 * (C) 2002-2004 Netfilter Core Team <coreteam@netfilter.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */

/*
 * 23 Apr 2001: Harald Welte <laforge@gnumonks.org>
 *    - new API and handling of conntrack/nat helpers
 *    - now capable of multiple expectations for one master
 * */

#include <linux/config.h>
#include <linux/types.h>
#include <linux/icmpv6.h>
#include <linux/ipv6.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv6.h>
#include <linux/module.h>
```

```
#include <linux/skbuff.h>
#include <linux/proc_fs.h>
#include <net/checksum.h>
#include <linux/spinlock.h>

#define IPV6_HDR_LEN    (sizeof(struct ipv6hdr))
#define IPV6_OPTHDR_LEN (sizeof(struct ipv6_opt_hdr))

#define ASSERT_READ_LOCK(x) MUST_BE_READ_LOCKED(&ip6_nat_lock)
#define ASSERT_WRITE_LOCK(x) MUST_BE_WRITE_LOCKED(&ip6_nat_lock)

#include <linux/netfilter_ipv6/ip6_nat.h>
#include <linux/netfilter_ipv6/ip6_nat_rule.h>
#include <linux/netfilter_ipv6/ip6_nat_protocol.h>
#include <linux/netfilter_ipv6/ip6_nat_core.h>
#include <linux/netfilter_ipv6/ip6_nat_helper.h>
#include <linux/netfilter_ipv6/ip6_tables.h>
#include <linux/netfilter_ipv6/ip6_conntrack_core.h>
#include <linux/netfilter_ipv4/listhelp.h>

#if 0
#define DEBUGP printk
#else
#define DEBUGP(format, args...)
#endif

#define   HOOKNAME(hooknum)   ((hooknum)   ==   NF_IP6_POST_ROUTING   ?
"POST_ROUTING"  \
                     : ((hooknum) == NF_IP6_PRE_ROUTING ? "PRE_ROUTING"
\
                        :  ((hooknum)  ==  NF_IP6_LOCAL_OUT  ?  "LOCAL_OUT"
\
                           :  ((hooknum)  ==  NF_IP6_LOCAL_IN  ?  "LOCAL_IN"
\
                            : "*ERROR*")))

static inline int call_expect(struct ip6_conntrack *master,
                        struct sk_buff **pskb,
                        unsigned int hooknum,
                        struct ip6_conntrack *ct,
                        struct ip6_nat_info *info)
{
      return master->nat.info.helper->expect(pskb, hooknum, ct, info);
}

static unsigned int
ip6_nat_fn(unsigned int hooknum,
        struct sk_buff **pskb,
        const struct net_device *in,
        const struct net_device *out,
        int (*okfn)(struct sk_buff *),
         unsigned int dataoff)
{

      struct ip6_conntrack *ct;
      enum ip6_conntrack_info ctinfo;
      struct ip6_nat_info *info;
      /* maniptype == SRC for postrouting. */

      enum ip6_nat_manip_type maniptype = HOOK2MANIP(hooknum);

      /* We never see fragments: conntrack defrags on pre-routing
         and local-out, and ip_nat_out protects post-routing. */

      IP6_NF_ASSERT(!((*pskb)->nh.ipv6h->frag_off
                   & htons(IP6_MF|IP6_OFFSET)));

      (*pskb)->nfcache |= NFC_UNKNOWN;

      /* If we had a hardware checksum before, it's now invalid */
      if ((*pskb)->ip_summed == CHECKSUM_HW){
            (*pskb)->ip_summed = CHECKSUM_NONE;
      }
      ct = ip6_conntrack_get(*pskb, &ctinfo);
      /* Can't track?  It's not due to stress, or conntrack would
         have dropped it.  Hence it's the user's responsibilty to
         packet filter it out, or implement conntrack/NAT for that
         protocol. 8) --RR */

      if (!ct) {

            return NF_ACCEPT;
      }

      switch (ctinfo) {
      case IP6_CT_RELATED:

      case IP6_CT_RELATED+IP6_CT_IS_REPLY:


            if ((*pskb)->nh.ipv6h->nexthdr == IPPROTO_ICMPV6) {

                  if (!icmpv6_reply_translation(pskb, ct, hooknum,
                                          CTINFO2DIR(ctinfo))){

                        return NF_DROP;
                  }
                  else{

                        return NF_ACCEPT;
                  }
            }

            /* Fall thru... (Only ICMPs can be IP_CT_IS_REPLY) */
      case IP6_CT_NEW:

            info = &ct->nat.info;

            WRITE_LOCK(&ip6_nat_lock);
            /* Seen it before?  This can happen for loopback, retrans,
               or local packets.. */

            if (!(info->initialized & (1 << maniptype))
#ifndef CONFIG_IP6_NF_NAT_LOCAL
```

```
                /* If this session has already been confirmed we must
not
                 * touch it again even if there is no mapping set up.
                 * Can only happen on local->local traffic with
                 * CONFIG_IP6_NF_NAT_LOCAL disabled.
                 */
                && !(ct->status & IPS_CONFIRMED)
#endif
                ) {
                  unsigned int ret;

                  if (ct->master
                      && master_ct6(ct)->nat.info.helper
                      && master_ct6(ct)->nat.info.helper->expect) {

                        ret = call_expect(master_ct6(ct), pskb,
                                      hooknum, ct, info);
                  } else {
#ifdef CONFIG_IP6_NF_NAT_LOCAL
                        /* LOCAL_IN hook doesn't have a chain!  */
                        if (hooknum == NF_IP6_LOCAL_IN){
                          ret = ip6_alloc_null_binding(ct, info,
                                                 hooknum);
}
                        else
#endif
                        ret = ip6_nat_rule_find(pskb, hooknum, in, out,
                                           ct, info);
                  }
                  if (ret != NF_ACCEPT) {

                        WRITE_UNLOCK(&ip6_nat_lock);
                        return ret;
                  }
            } else
                  DEBUGP("Already setup manip %s for ct %p\n",
                         maniptype == IP6_NAT_MANIP_SRC ? "SRC" :
"DST",
                         ct);

            WRITE_UNLOCK(&ip6_nat_lock);
            break;

      default:
            /* ESTABLISHED */

            IP6_NF_ASSERT(ctinfo == IP6_CT_ESTABLISHED
                       ||                   ctinfo                    ==
(IP6_CT_ESTABLISHED+IP6_CT_IS_REPLY));
            info = &ct->nat.info;
      }
      IP6_NF_ASSERT(info);
      return ip6_do_bindings(ct, ctinfo, info, hooknum, pskb, dataoff);
}

static unsigned int
ip6_nat_out(unsigned int hooknum,
         struct sk_buff **pskb,
         const struct net_device *in,
         const struct net_device *out,
         int (*okfn)(struct sk_buff *),
          unsigned int dataoff)
{
      /* root is playing with raw sockets. */
      if ((*pskb)->len < sizeof(struct ipv6hdr)
          || IPV6_HDR_LEN < sizeof(struct ipv6hdr)){
            return NF_ACCEPT;
}
      /* We can hit fragment here; forwarded packets get
         defragmented by connection tracking coming in, then
         fragmented (grr) by the forward code.

         In future: If we have nfct != NULL, AND we have NAT
         initialized, AND there is no helper, then we can do full
         NAPT on the head, and IP-address-only NAT on the rest.

         I'm starting to have nightmares about fragments.  */


      /*if ((*pskb)->nh.ipv6h->fh & htons(IP6_MF|IP6_OFFSET)) {
            *pskb = ip6_ct_gather_frags(*pskb);

            if (!*pskb)
                  return NF_STOLEN;
                  }*/
      return ip6_nat_fn(hooknum, pskb, in, out, okfn, dataoff);
}

#ifdef CONFIG_IP6_NF_NAT_LOCAL
static unsigned int
ip6_nat_local_fn(unsigned int hooknum,
            struct sk_buff **pskb,
            const struct net_device *in,
            const struct net_device *out,
            int (*okfn)(struct sk_buff *),
             unsigned int dataoff)
{
      struct in6_addr saddr, daddr;
      unsigned int ret;

      /* root is playing with raw sockets. */
      if ((*pskb)->len < sizeof(struct ipv6hdr)
          || IPV6_HDR_LEN < sizeof(struct ipv6hdr))
            return NF_ACCEPT;

      saddr = (*pskb)->nh.ipv6h->saddr;
      daddr = (*pskb)->nh.ipv6h->daddr;

      ret = ip6_nat_fn(hooknum, pskb, in, out, okfn, dataoff);
      /*if (ret != NF_DROP && ret != NF_STOLEN
          && ((*pskb)->nh.ipv6h->saddr != saddr
            || (*pskb)->nh.ipv6h->daddr != daddr))
            return ip6_route_me_harder(pskb) == 0 ? ret : NF_DROP;*/
      return ret;
```

```
}
#endif

/* We must be after connection tracking and before packet filtering. */

/* Before packet filtering, change destination */
static struct nf_hook_ops ip6_nat_in_ops = {
      .hook       = ip6_nat_fn,
      .owner            = THIS_MODULE,
      .pf         = PF_INET6,
      .hooknum    = NF_IP6_PRE_ROUTING,
      .priority   = NF_IP6_PRI_NAT_DST,
};

/* After packet filtering, change source */
static struct nf_hook_ops ip6_nat_out_ops = {
      .hook       = ip6_nat_out,
      .owner            = THIS_MODULE,
      .pf         = PF_INET6,
      .hooknum    = NF_IP6_POST_ROUTING,
      .priority   = NF_IP6_PRI_NAT_SRC,
};

#ifdef CONFIG_IP6_NF_NAT_LOCAL
/* Before packet filtering, change destination */
static struct nf_hook_ops ip6_nat_local_out_ops = {
      .hook       = ip6_nat_local_fn,
      .owner            = THIS_MODULE,
      .pf         = PF_INET6,
      .hooknum    = NF_IP6_LOCAL_OUT,
      .priority   = NF_IP6_PRI_NAT_DST,
};

/* After packet filtering, change source for reply packets of LOCAL_OUT
DNAT */
static struct nf_hook_ops ip6_nat_local_in_ops = {
      .hook       = ip6_nat_fn,
      .owner            = THIS_MODULE,
      .pf         = PF_INET6,
      .hooknum    = NF_IP6_LOCAL_IN,
      .priority   = NF_IP6_PRI_NAT_SRC,
};
#endif

/* Protocol registration. */
int ip6_nat_protocol_register(struct ip6_nat_protocol *proto)
{
      int ret = 0;
      struct list_head *i;

      WRITE_LOCK(&ip6_nat_lock);
      list_for_each(i, &ip6_protos) {
            if (((struct ip6_nat_protocol *)i)->protonum
                == proto->protonum) {
                  ret = -EBUSY;
                  goto out;
            }
      }

      list_prepend(&ip6_protos, proto);
 out:
      WRITE_UNLOCK(&ip6_nat_lock);
      return ret;
}

/* Noone stores the protocol anywhere; simply delete it. */
void ip6_nat_protocol_unregister(struct ip6_nat_protocol *proto)
{
      WRITE_LOCK(&ip6_nat_lock);
      LIST_DELETE(&ip6_protos, proto);
      WRITE_UNLOCK(&ip6_nat_lock);

      /* Someone could be still looking at the proto in a bh. */
      synchronize_net();
}

static int init_or_cleanup(int init)
{
      int ret = 0;

      need_ip6_conntrack();

      if (!init) goto cleanup;

      ret = ip6_nat_rule_init();
      if (ret < 0) {
            printk("ip6_nat_init: can't setup rules.\n");
            goto cleanup_nothing;
      }
      ret = ip6_nat_init();
      if (ret < 0) {
            printk("ip6_nat_init: can't setup rules.\n");
            goto cleanup_rule_init;
      }
      ret = nf_register_hook(&ip6_nat_in_ops);
      if (ret < 0) {
            printk("ip6_nat_init: can't register in hook.\n");
            goto cleanup_nat;
      }
      ret = nf_register_hook(&ip6_nat_out_ops);
      if (ret < 0) {
            printk("ip6_nat_init: can't register out hook.\n");
            goto cleanup_inops;
      }
#ifdef CONFIG_IP6_NF_NAT_LOCAL
      ret = nf_register_hook(&ip6_nat_local_out_ops);
      if (ret < 0) {
            printk("ip6_nat_init: can't register local out hook.\n");
            goto cleanup_outops;
      }
      ret = nf_register_hook(&ip6_nat_local_in_ops);
      if (ret < 0) {
            printk("ip6_nat_init: can't register local in hook.\n");
            goto cleanup_localoutops;
      }
#endif
      return ret;

 cleanup:
#ifdef CONFIG_IP6_NF_NAT_LOCAL
      nf_unregister_hook(&ip6_nat_local_in_ops);
 cleanup_localoutops:
      nf_unregister_hook(&ip6_nat_local_out_ops);
 cleanup_outops:
#endif
      nf_unregister_hook(&ip6_nat_out_ops);
 cleanup_inops:
      nf_unregister_hook(&ip6_nat_in_ops);
 cleanup_nat:
      ip6_nat_cleanup();
 cleanup_rule_init:
      ip6_nat_rule_cleanup();
 cleanup_nothing:
      MUST_BE_READ_WRITE_UNLOCKED(&ip6_nat_lock);
      return ret;
}

static int __init init(void)
{
      return init_or_cleanup(1);
}

static void __exit fini(void)
{
      init_or_cleanup(0);
}

module_init(init);
module_exit(fini);

EXPORT_SYMBOL(ip6_nat_setup_info);
EXPORT_SYMBOL(ip6_nat_protocol_register);
EXPORT_SYMBOL(ip6_nat_protocol_unregister);
EXPORT_SYMBOL(ip6_nat_helper_register);
EXPORT_SYMBOL(ip6_nat_helper_unregister);
EXPORT_SYMBOL(ip6_nat_cheat_check);
/*
EXPORT_SYMBOL(ip_nat_cheat_check);
*/
EXPORT_SYMBOL(ip6_nat_mangle_tcp_packet);
EXPORT_SYMBOL(ip6_nat_mangle_udp_packet);
EXPORT_SYMBOL(ip6_nat_used_tuple);
MODULE_LICENSE("GPL");
```

/INCLUDE/LINUX/NETFILTER_IPV6/IP6T_IPRANGE.H

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: include/linux/ ip6t_iprange.h
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */


#ifndef _IP6T_IPRANGE_H
#define _IP6T_IPRANGE_H

#define IPRANGE_SRC           0x01  /* Match source IP address */
#define IPRANGE_DST           0x02  /* Match destination IP address */
#define IPRANGE_SRC_INV       0x10  /* Negate the condition */
#define IPRANGE_DST_INV       0x20  /* Negate the condition */

struct ip6t_iprange {
      /* Inclusive: network order. */
      struct in6_addr min_ip, max_ip;
};

struct ip6t_iprange_info
{
      struct ip6t_iprange src;
      struct ip6t_iprange dst;

      /* Flags from above */
      u_int8_t flags;
};

#endif /* _IP6T_IPRANGE_H */
```

/NET/IPV6/NETFILTER/IP6T_NETMAP.C

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: net/ipv4/netfilter/ipt_NETMAP.c
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

/* NETMAP - static NAT mapping of IP network addresses (1:1).
 * The mapping can be applied to source (POSTROUTING),
 * destination (PREROUTING), or both (with separate rules).
 */

/* (C) 2000-2001 Svenning Soerensen <svenning@post5.tele.dk>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */

#include <linux/config.h>
#include <linux/ipv6.h>
#include <linux/module.h>
#include <linux/netdevice.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv6.h>
#include <linux/netfilter_ipv6/ip6_nat_rule.h>

#define MODULENAME "NETMAP"
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Svenning Soerensen <svenning@post5.tele.dk>");
MODULE_DESCRIPTION("iptables 1:1 NAT mapping of IP networks target");

#if 0
#define DEBUGP printk
```

```
#else
#define DEBUGP(format, args...)
#endif

static int
check(const char *tablename,
      const struct ip6t_entry *e,
      void *targinfo,
      unsigned int targinfosize,
      unsigned int hook_mask)
{
      const struct ip6_nat_multi_range *mr = targinfo;

      if (strcmp(tablename, "nat") != 0) {
            DEBUGP(MODULENAME":check: bad table `%s'.\n", tablename);
            return 0;
      }
      if (targinfosize != IP6T_ALIGN(sizeof(*mr))) {
            DEBUGP(MODULENAME":check: size %u.\n", targinfosize);
            return 0;
      }
      if   (hook_mask   &   ~((1   <<   NF_IP6_PRE_ROUTING)   |   (1   <<
NF_IP6_POST_ROUTING))) {
            DEBUGP(MODULENAME":check: bad hooks %x.\n", hook_mask);
            return 0;
      }
      if (!(mr->range[0].flags & IP6_NAT_RANGE_MAP_IPS)) {
            DEBUGP(MODULENAME":check: bad MAP_IPS.\n");
            return 0;
      }
      if (mr->rangesize != 1) {
            DEBUGP(MODULENAME":check:   bad   rangesize   %u.\n",   mr-
>rangesize);
            return 0;
      }
      return 1;
}

static unsigned int
target(struct sk_buff **pskb,
       const struct net_device *in,
       const struct net_device *out,
       unsigned int hooknum,
       const void *targinfo,
       void *userinfo)
{
      struct ip6_conntrack *ct;
      enum ip6_conntrack_info ctinfo;
      struct in6_addr new_ip, netmask;
      const struct ip6_nat_multi_range *mr = targinfo;
      struct ip6_nat_multi_range newrange;

      IP6_NF_ASSERT(hooknum == NF_IP6_PRE_ROUTING
                 || hooknum == NF_IP6_POST_ROUTING);
      ct = ip6_conntrack_get(*pskb, &ctinfo);
```

```
	netmask.s6_addr[0]   =   ~(mr->range[0].min_ip.s6_addr[0]   ^   mr-
>range[0].max_ip.s6_addr[0]);
	netmask.s6_addr[1]   =   ~(mr->range[0].min_ip.s6_addr[1]   ^   mr-
>range[0].max_ip.s6_addr[1]);
	netmask.s6_addr[2]   =   ~(mr->range[0].min_ip.s6_addr[2]   ^   mr-
>range[0].max_ip.s6_addr[2]);
	netmask.s6_addr[3]   =   ~(mr->range[0].min_ip.s6_addr[3]   ^   mr-
>range[0].max_ip.s6_addr[3]);

	if (hooknum == NF_IP6_PRE_ROUTING){
		new_ip.s6_addr[0]  =  (*pskb)->nh.ipv6h->daddr.s6_addr[0]  &
~netmask.s6_addr[0];
		new_ip.s6_addr[1]  =  (*pskb)->nh.ipv6h->daddr.s6_addr[1]  &
~netmask.s6_addr[1];
		new_ip.s6_addr[2]  =  (*pskb)->nh.ipv6h->daddr.s6_addr[2]  &
~netmask.s6_addr[2];
		new_ip.s6_addr[3]  =  (*pskb)->nh.ipv6h->daddr.s6_addr[3]  &
~netmask.s6_addr[3];
	}
	else{
		new_ip.s6_addr[0]  =  (*pskb)->nh.ipv6h->saddr.s6_addr[0]  &
~netmask.s6_addr[0];
		new_ip.s6_addr[1]  =  (*pskb)->nh.ipv6h->saddr.s6_addr[1]  &
~netmask.s6_addr[1];
		new_ip.s6_addr[2]  =  (*pskb)->nh.ipv6h->saddr.s6_addr[2]  &
~netmask.s6_addr[2];
		new_ip.s6_addr[3]  =  (*pskb)->nh.ipv6h->saddr.s6_addr[3]  &
~netmask.s6_addr[3];
	}

	new_ip.s6_addr[0]     |=     mr->range[0].min_ip.s6_addr[0]     &
netmask.s6_addr[0];
	new_ip.s6_addr[1]     |=     mr->range[0].min_ip.s6_addr[1]     &
netmask.s6_addr[1];
	new_ip.s6_addr[2]     |=     mr->range[0].min_ip.s6_addr[2]     &
netmask.s6_addr[2];
	new_ip.s6_addr[3]     |=     mr->range[0].min_ip.s6_addr[3]     &
netmask.s6_addr[3];

	newrange = ((struct ip6_nat_multi_range)
	{ 1, { { mr->range[0].flags | IP6_NAT_RANGE_MAP_IPS,
		 new_ip, new_ip,
		 mr->range[0].min, mr->range[0].max } } });

	/* Hand modified range to generic setup. */
	return ip6_nat_setup_info(ct, &newrange, hooknum);
}

static struct ip6t_target target_module = {
	.name		= MODULENAME,
	.target 	= target,
	.checkentry	= check,
	.me		= THIS_MODULE
};

static int __init init(void)
{
	return ip6t_register_target(&target_module);
}

static void __exit fini(void)
{
	ip6t_unregister_target(&target_module);
}

module_init(init);
module_exit(fini);
```

/NET/IPV6/NETFILTER/IP6T_SAME.C

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: net/ipv4/netfilter/ipt_SAME.c
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

/* Same.  Just like SNAT, only try to make the connections
 *      between client A and server B always have the same source ip.
 *
 * (C) 2000 Paul `Rusty' Russell
 * (C) 2001 Martin Josefsson
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 *
 * 010320 Martin Josefsson <gandalf@wlug.westbo.se>
 *    * copied ipt_BALANCE.c to ipt_SAME.c and changed a few things.
 * 010728 Martin Josefsson <gandalf@wlug.westbo.se>
 *    * added --nodst to not include destination-ip in new source
 *      calculations.
 *    * added some more sanity-checks.
 * 010729 Martin Josefsson <gandalf@wlug.westbo.se>
 *    * fixed a buggy if-statement in same_check(), should have
 *      used ntohl() but didn't.
 *    * added support for multiple ranges. IPT_SAME_MAX_RANGE is
 *      defined in linux/include/linux/netfilter_ipv4/ipt_SAME.h
 *      and is currently set to 10.
 *    * added support for 1-address range, nice to have now that
 *      we have multiple ranges.
 */
#include <linux/types.h>
#include <linux/ipv6.h>
```

```
#include <linux/timer.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netdevice.h>
#include <linux/if.h>
#include <linux/inetdevice.h>
#include <net/protocol.h>
#include <net/checksum.h>
#include <linux/netfilter_ipv6.h>
#include <linux/netfilter_ipv6/ip6_nat_rule.h>
#include <linux/netfilter_ipv6/ip6t_SAME.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Martin Josefsson <gandalf@wlug.westbo.se>");
MODULE_DESCRIPTION("iptables special SNAT module for consistent
sourceip");

#if 1
#define DEBUGP printk
#else
#define DEBUGP(format, args...)
#endif

static int
same_check(const char *tablename,
            const struct ip6t_entry *e,
            void *targinfo,
            unsigned int targinfosize,
            unsigned int hook_mask)
{
      unsigned int count, countess0, countess1, countess2, countess3,
rangeip, index = 0;
      struct ip6t_same_info *mr = targinfo;

      mr->ipnum = 0;

      if (strcmp(tablename, "nat") != 0) {
            DEBUGP("same_check: bad table `%s'.\n", tablename);
            return 0;
      }
      if (targinfosize != IP6T_ALIGN(sizeof(*mr))) {
            DEBUGP("same_check: size %u.\n", targinfosize);
            return 0;
      }
      if (hook_mask & ~(1 << NF_IP6_PRE_ROUTING | 1 <<
NF_IP6_POST_ROUTING)) {
            DEBUGP("same_check: bad hooks %x.\n", hook_mask);
            return 0;
      }
      if (mr->rangesize < 1) {
            DEBUGP("same_check: need at least one dest range.\n");
            return 0;
      }
      if (mr->rangesize > IP6T_SAME_MAX_RANGE) {
            DEBUGP("same_check: too many ranges specified, maximum "
                        "is %u ranges\n",
                        IP6T_SAME_MAX_RANGE);
            return 0;
      }
      for (count = 0; count < mr->rangesize; count++) {
            if(      (ntohl(mr->range[count].min_ip.s6_addr32[0])      >
ntohl(mr->range[count].max_ip.s6_addr32[0]))

                ||    ((ntohl(mr->range[count].min_ip.s6_addr32[0])    ==
ntohl(mr->range[count].max_ip.s6_addr32[0]))
                  &&    (ntohl(mr->range[count].min_ip.s6_addr32[1])    >
ntohl(mr->range[count].max_ip.s6_addr32[1])))

                ||    ((ntohl(mr->range[count].min_ip.s6_addr32[0])    ==
ntohl(mr->range[count].max_ip.s6_addr32[0]))
                  &&   (ntohl(mr->range[count].min_ip.s6_addr32[1])   ==
ntohl(mr->range[count].max_ip.s6_addr32[1]))
                  &&    (ntohl(mr->range[count].min_ip.s6_addr32[2])    >
ntohl(mr->range[count].max_ip.s6_addr32[2])))

                ||    ((ntohl(mr->range[count].min_ip.s6_addr32[0])    ==
ntohl(mr->range[count].max_ip.s6_addr32[0]))
                  &&   (ntohl(mr->range[count].min_ip.s6_addr32[1])   ==
ntohl(mr->range[count].max_ip.s6_addr32[1]))
                  &&   (ntohl(mr->range[count].min_ip.s6_addr32[2])   ==
ntohl(mr->range[count].max_ip.s6_addr32[2]))
                  &&    (ntohl(mr->range[count].min_ip.s6_addr32[3])    >
ntohl(mr->range[count].max_ip.s6_addr32[3]))))
              {
                DEBUGP("same_check: min_ip is larger than max_ip in "
                     "range         `          %x:%x:%x:%x:%x:%x:%x:%x-
%x:%x:%x:%x:%x:%x:%x:%x'.\n",
                     NIP6(mr->range[count].min_ip),
                     NIP6(mr->range[count].max_ip));
                return 0;
              }
            if (!(mr->range[count].flags & IP6_NAT_RANGE_MAP_IPS)) {
              DEBUGP("same_check: bad MAP_IPS.\n");
              return 0;
            }
            rangeip  =  (((ntohl(mr->range[count].max_ip.s6_addr32[0])  -
ntohl(mr->range[count].min_ip.s6_addr32[0])) + 1) *
                     ((ntohl(mr->range[count].max_ip.s6_addr32[1])     -
ntohl(mr->range[count].min_ip.s6_addr32[1])) + 1) *
                     ((ntohl(mr->range[count].max_ip.s6_addr32[2])     -
ntohl(mr->range[count].min_ip.s6_addr32[2])) + 1) *
                     ((ntohl(mr->range[count].max_ip.s6_addr32[3])     -
ntohl(mr->range[count].min_ip.s6_addr32[3])) + 1));
            mr->ipnum += rangeip;
            DEBUGP("same_check:   range   %u,   ipnum   =   %u\n",   count,
rangeip);
      }

      DEBUGP("same_check: total ipaddresses = %u\n", mr->ipnum);
      mr->iparray  =  kmalloc((sizeof(struct  in6_addr)  *  mr->ipnum),
GFP_KERNEL);
      if (!mr->iparray) {
        DEBUGP("same_check: Couldn't allocate %u bytes "
             "for %u ipaddresses!\n",
             (sizeof(struct in6_addr) * mr->ipnum), mr->ipnum);
        return 0;
      }
      DEBUGP("same_check: Allocated %u bytes for %u ipaddresses.\n",
             (sizeof(struct in6_addr) * mr->ipnum), mr->ipnum);

      for (count = 0; count < mr->rangesize; count++) {

        for (countess0 = ntohl(mr->range[count].min_ip.s6_addr32[0]);
             countess0 <= ntohl(mr->range[count].max_ip.s6_addr32[0]);
             countess0++) {

          countess1 = 0;
          for (countess1 = ntohl(mr->range[count].min_ip.s6_addr32[1]);
             countess1 <= ntohl(mr->range[count].max_ip.s6_addr32[1]);
             countess1++) {

            countess2 = 0;
            for             (countess2             =            ntohl(mr-
>range[count].min_ip.s6_addr32[2]);
               countess2                  <=                 ntohl(mr-
>range[count].max_ip.s6_addr32[2]);
               countess2++) {

            countess3 = 0;
            for             (countess3             =            ntohl(mr-
>range[count].min_ip.s6_addr32[3]);
                 countess3                 <=                ntohl(mr-
>range[count].max_ip.s6_addr32[3]);
                 countess3++) {

                  mr->iparray[index].s6_addr32[0] = countess0;
                  mr->iparray[index].s6_addr32[1] = countess1;
                  mr->iparray[index].s6_addr32[2] = countess2;
                  mr->iparray[index].s6_addr32[3] = countess3;

                  DEBUGP("same_check:      Added      ipaddress       `
%x:%x:%x:%x:%x:%x:%x:%x' "
                        "in index %u.\n",
                        NIP6(mr->iparray[index]), index);
                  index++;
            }
            }
          }
        }
      }

      return 1;
}

static void
same_destroy(void *targinfo,
            unsigned int targinfosize)
{
      struct ip6t_same_info *mr = targinfo;

      kfree(mr->iparray);
      DEBUGP("same_destroy: Deallocated %u bytes for %u
ip6addresses.\n",
                  (sizeof(struct in6_addr) * mr->ipnum), mr->ipnum);
}

static unsigned int
same_target(struct sk_buff **pskb,
            const struct net_device *in,
            const struct net_device *out,
            unsigned int hooknum,
            const void *targinfo,
            void *userinfo)
{
      struct ip6_conntrack *ct;
      enum ip6_conntrack_info ctinfo;
      struct in6_addr tmpip, new_ip;
      u_int32_t aindex;
      const struct ip6t_same_info *mr = targinfo;
      struct ip6_nat_multi_range newrange;
      const struct ip6_conntrack_tuple *t;

      IP6_NF_ASSERT(hooknum == NF_IP6_PRE_ROUTING ||
                  hooknum == NF_IP6_POST_ROUTING);
      ct = ip6_conntrack_get(*pskb, &ctinfo);

      t = &ct->tuplehash[IP6_CT_DIR_ORIGINAL].tuple;

      /* Base new source on real src ip and optionally dst ip,
         giving some hope for consistency across reboots.
         Here we calculate the index in mr->iparray which
         holds the ipaddress we should use */

      tmpip.s6_addr32[0] = ntohl(t->src.ip.s6_addr32[0]);
      tmpip.s6_addr32[1] = ntohl(t->src.ip.s6_addr32[1]);
      tmpip.s6_addr32[2] = ntohl(t->src.ip.s6_addr32[2]);
      tmpip.s6_addr32[3] = ntohl(t->src.ip.s6_addr32[3]);

      if (!(mr->info & IP6T_SAME_NODST)){
        tmpip.s6_addr32[0] += ntohl(t->dst.ip.s6_addr32[0]);
        tmpip.s6_addr32[1] += ntohl(t->dst.ip.s6_addr32[1]);
        tmpip.s6_addr32[2] += ntohl(t->dst.ip.s6_addr32[2]);
        tmpip.s6_addr32[3] += ntohl(t->dst.ip.s6_addr32[3]);
      }

      aindex = ((tmpip.s6_addr32[0] + tmpip.s6_addr32[1] +
tmpip.s6_addr32[2] + tmpip.s6_addr32[3]) % mr->ipnum);

      new_ip.s6_addr32[0] = ntohl(mr->iparray[aindex].s6_addr32[0]);
      new_ip.s6_addr32[1] = htonl(mr->iparray[aindex].s6_addr32[1]);
      new_ip.s6_addr32[2] = htonl(mr->iparray[aindex].s6_addr32[2]);
      new_ip.s6_addr32[3] = htonl(mr->iparray[aindex].s6_addr32[3]);

      DEBUGP("ip6t_SAME: src= %x:%x:%x:%x:%x:%x:%x:%x dst=
%x:%x:%x:%x:%x:%x:%x:%x, "
                  "new src= %x:%x:%x:%x:%x:%x:%x:%x\n",
                  NIP6(t->src.ip), NIP6(t->dst.ip),
                  NIP6(new_ip));

      /* Transfer from original range. */
      newrange = ((struct ip6_nat_multi_range)
            { 1, { { mr->range[0].flags | IP6_NAT_RANGE_MAP_IPS,
                   new_ip, new_ip,
                   mr->range[0].min, mr->range[0].max } } });

      /* Hand modified range to generic setup. */
      return ip6_nat_setup_info(ct, &newrange, hooknum);
}

static struct ip6t_target same_reg = {
      .name       = "SAME",
      .target           = same_target,
      .checkentry = same_check,
      .destroy    = same_destroy,
      .me         = THIS_MODULE,
};

static int __init init(void)
{
      return ip6t_register_target(&same_reg);
}

static void __exit fini(void)
{
      ip6t_unregister_target(&same_reg);
}

module_init(init);
module_exit(fini);
```

/INCLUDE/LINUX/NETFILTER_IPV6/IP6T_SAME.H

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: include/linux/ipt_SAME.h
 *
 * Created by:
 *      Trevor J. Baumgartner
 *      Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */


#ifndef _IP6T_SAME_H
#define _IP6T_SAME_H

#define IP6T_SAME_MAX_RANGE   10

#define IP6T_SAME_NODST       0x01

struct ip6t_same_info
{
      unsigned char info;
      u_int32_t rangesize;
      u_int32_t ipnum;
      struct in6_addr *iparray;

      /* hangs off end. */
      struct ip6_nat_range range[IP6T_SAME_MAX_RANGE];
};

#endif /*_IP6T_SAME_H*/
```

/NET/CORE/NETFILTER.C

```
/* netfilter.c: look after the filters for various protocols.
 * Heavily influenced by the old firewall.c by David Bonn and Alan Cox.
 *
 * Thanks to Rob `CmdrTaco' Malda for not influencing this code in any
 * way.
 *
 * Rusty Russell (C)2000 -- This code is GPL.
 *
 *  February 2000: Modified by James Morris to have 1 queue per
protocol.
 * 15-Mar-2000:   Added NF_REPEAT --RR.
 * 24-May-2004:   Added ip6_skb_make_writable() - TB MP
 */


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * *
 * TB MP - This funciton is the same as skb_ip_make_writable except
variables and function
 * names are updated to reflect changes made in the IPv6 suite.  This
function is necessary
 * for NAT, or any other function, to write to the skb.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * */

int skb_ip6_make_writable(struct sk_buff **pskb, unsigned int
writable_len)
{
      struct sk_buff *nskb;
      unsigned int iplen;

      if (writable_len > (*pskb)->len)
            return 0;

      /* Not exclusive use of packet?  Must copy. */
      if (skb_shared(*pskb) || skb_cloned(*pskb))
            goto copy_skb;

      /* Alexey says IP hdr is always modifiable and linear, so ok. */
      if (writable_len <= IPV6_HDR_LEN)
            return 1;

      iplen = writable_len - IPV6_HDR_LEN;

      /* DaveM says protocol headers are also modifiable. */
      switch ((*pskb)->nh.ipv6h->nexthdr) {
      case IPPROTO_TCP: {
            struct tcphdr hdr;
            if (skb_copy_bits(*pskb, IPV6_HDR_LEN,
                          &hdr, sizeof(hdr)) != 0)
              goto copy_skb;
            if (writable_len <= (IPV6_HDR_LEN + hdr.doff*4))
              goto pull_skb;
            goto copy_skb;
      }
      case IPPROTO_UDP:
            if (writable_len <= IPV6_HDR_LEN + sizeof(struct udphdr))
                  goto pull_skb;
            goto copy_skb;
      case IPPROTO_ICMPV6:
            if (writable_len
                <= IPV6_HDR_LEN + sizeof(struct icmp6hdr))
                  goto pull_skb;
            goto copy_skb;
      /* Insert other cases here as desired */
      }

copy_skb:
      nskb = skb_copy(*pskb, GFP_ATOMIC);
      if (!nskb)
            return 0;
      BUG_ON(skb_is_nonlinear(nskb));

      /* Rest of kernel will get very unhappy if we pass it a
         suddenly-orphaned skbuff */
      if ((*pskb)->sk)
            skb_set_owner_w(nskb, (*pskb)->sk);
      kfree_skb(*pskb);
      *pskb = nskb;
      return 1;

pull_skb:
      return pskb_may_pull(*pskb, writable_len);
}
EXPORT_SYMBOL(skb_ip6_make_writable);

/* TB MP - END NAT CODE*/
```

/HOME/IPTABLES-1.2.9RC1/EXTENSIONS/LIBIP6T_SNAT.C

```
/*
 * IPv6 Network Address Translation
 * Linux INET6 Implementation
 *
 * Created based on: /home/iptables-1.2.9rc1/extensions/libipt_SNAT.c
 *
 * Created by:
 *	Trevor J. Baumgartner
 *	Matthew D. W. Phillips
 *
 *
 *
 * Except where noted, porting involved rote updates of function names
 * and datatypes to reflect those being used in IPv6 versus those being
 * used in IPv4.  For example, instead of using an unsigned 32 bit
 * integer for the IPv4 address, an in6_addr struct is used for IPv6,
 * or instead of using the pointer 'icmphdr' to access the icmp header,
 * IPv6 uses 'icmp6hdr'.  Substantial changes are explained in detail.
 *
 * Certain areas necessitated breaking the IPv6 address down into array
 * format in order to perform binary operations on the address.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

/* Shared library add-on to iptables to add source-NAT support. */
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <ctype.h>
#include <stdarg.h>
#include <limits.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <getopt.h>
#include <ip6tables.h>
#include <linux/netfilter_ipv6/ip6_tables.h>
#include <linux/netfilter_ipv6/ip6_nat_rule.h>

static char *
addr_to_numeric(const struct in6_addr *addrp)
{
	/* 0000:0000:0000:0000:0000:000.000.000.000
	 * 0000:0000:0000:0000:0000:0000:0000:0000 */
	static char buf[50+1];
	return (char *)inet_ntop(AF_INET6, addrp, buf, sizeof(buf));
```

```
}

static struct in6_addr *
numeric_to_addr(const char *num)
{
      static struct in6_addr ap;
      int err;
      if ((err=inet_pton(AF_INET6, num, &ap)) == 1)
            return &ap;
#ifdef DEBUG
      fprintf(stderr, "\nnumeric2addr: %d\n", err);
#endif
      return (struct in6_addr *)NULL;
}

      /* Source NAT data consists of a multi-range, indicating where to
      map to. */

struct ip6t_natinfo
{
      struct ip6t_entry_target t;
      struct ip6_nat_multi_range mr;
};

/* Function which prints out usage message. */
static void
help(void)
{
      printf(
"SNAT v%s options:\n"
" --to-source <ipaddr>[-<ipaddr>][:port-port]\n"
"                       Address to map source to.\n"
"                       (You can use this more than once)\n\n",
IPTABLES_VERSION);
}

static struct option opts[] = {
      { "to-source", 1, 0, '1' },
      { 0 }
};

/* Initialize the target. */
static void
init(struct ip6t_entry_target *t, unsigned int *nfcache)
{
      /* Can't cache this */
      *nfcache |= NFC_UNKNOWN;
}

static struct ip6t_natinfo *
      append_range(struct    ip6t_natinfo    *info,    const    struct
      ip6_nat_range *range)
{
      unsigned int size;

      /* One rangesize already in struct ipt_natinfo */
      size = IP6T_ALIGN(sizeof(*info) + info->mr.rangesize *
      sizeof(*range));

      info = realloc(info, size);
      if (!info)
            exit_error(OTHER_PROBLEM, "Out of memory\n");

      info->t.u.target_size = size;
      info->mr.range[info->mr.rangesize] = *range;
      info->mr.rangesize++;

      return info;
}

/* Ranges expected in network order. */
static struct ip6t_entry_target *
parse_to(char *arg, int portok, struct ip6t_natinfo *info)
{
      struct ip6_nat_range range;
      char *colon;
      memset(&range, 0, sizeof(range));
      colon = strchr(arg, '@');
      struct in6_addr *ip;


      /*TB MP - This section deals with ports. The scope of our work
      did not require port mappings or IP ranges therefore this section
      was commented out and is untested*/
      /*
        if(colon){
      if (!portok)
        exit_error(PARAMETER_PROBLEM,
                 "Need TCP or UDP with port specification");

      range.flags |= IP6_NAT_RANGE_PROTO_SPECIFIED;

      port = atoi(colon+1);
      if (port == 0 || port > 65535)
        exit_error(PARAMETER_PROBLEM,
                 "Port `%s' not valid\n", colon+1);

      dash = strchr(colon, '-');
      if (!dash) {
        range.min.tcp.port
          = range.max.tcp.port
          = htons(port);
      } else {
        int maxport;

        maxport = atoi(dash + 1);
        if (maxport == 0 || maxport > 65535)
          exit_error(PARAMETER_PROBLEM,
                   "Port `%s' not valid\n", dash+1);
        if (maxport < port)

          exit_error(PARAMETER_PROBLEM,
                   "Port range `%s' funky\n", colon+1);
        range.min.tcp.port = htons(port);
        range.max.tcp.port = htons(maxport);
      }

      if (colon == arg)
        return &(append_range(info, &range)->t);
      *colon = '\0';


      range.flags |= IP6_NAT_RANGE_MAP_IPS;
      dash = strchr(arg, '-');
      if (colon && dash && dash > colon)
            dash = NULL;

      if (dash)
            *dash = '\0';
      */
      ip = numeric_to_addr(arg);
      range.flags |= IP6_NAT_RANGE_MAP_IPS;
      if (!ip)
                  exit_error(PARAMETER_PROBLEM,    "Bad    IP    address
                  `%s'\n", arg);

      range.min_ip.s6_addr32[0] = ip->s6_addr32[0];
      range.min_ip.s6_addr32[1] = ip->s6_addr32[1];
      range.min_ip.s6_addr32[2] = ip->s6_addr32[2];
      range.min_ip.s6_addr32[3] = ip->s6_addr32[3];

      /*
      if (dash) {
            ip = dotted_to_addr(dash+1);
            if (!ip)
                        exit_error(PARAMETER_PROBLEM,  "Bad  IP  address
                        `%s'\n", dash+1
      } else{
      */

      return &(append_range(info, &range)->t);
}

/* Function which parses command options; returns true if it
   ate an option */
static int
parse(int c, char **argv, int invert, unsigned int *flags,
      const struct ip6t_entry *entry,
      struct ip6t_entry_target **target)
{
      struct ip6t_natinfo *info = (void *)*target;
      int portok;

      if (entry->ipv6.proto == IPPROTO_TCP
          || entry->ipv6.proto == IPPROTO_UDP)
            portok = 1;
      else
            portok = 0;

      switch (c) {
```

```
      case '1':
            if (check_inverse(optarg, &invert, NULL, 0))
                  exit_error(PARAMETER_PROBLEM,
                            "Unexpected `!' after --to-source");

            *target = parse_to(optarg, portok, info);
            *flags = 1;
            return 1;
      default:
            return 0;
      }
}

/* Final check; must have specfied --to-source. */
static void final_check(unsigned int flags)
{
      if (!flags)
            exit_error(PARAMETER_PROBLEM,
                      "You must specify --to-source");
}

static void print_range(const struct ip6_nat_range *r)
{
      if (r->flags & IP6_NAT_RANGE_MAP_IPS) {

            struct in6_addr a;
            a.s6_addr32[0] = r->min_ip.s6_addr32[0];
            a.s6_addr32[1] = r->min_ip.s6_addr32[1];
            a.s6_addr32[2] = r->min_ip.s6_addr32[2];
            a.s6_addr32[3] = r->min_ip.s6_addr32[3];
            printf("%s", addr_to_numeric(&a));
            if ((r->max_ip.s6_addr32[0] != r->min_ip.s6_addr32[0]) ||
                (r->max_ip.s6_addr32[1] != r->min_ip.s6_addr32[1]) ||
                (r->max_ip.s6_addr32[2] != r->min_ip.s6_addr32[2]) ||
                (r->max_ip.s6_addr32[3] != r->min_ip.s6_addr32[3])) {
                  a.s6_addr32[0] = r->max_ip.s6_addr32[0];
                  a.s6_addr32[1] = r->max_ip.s6_addr32[1];
                  a.s6_addr32[2] = r->max_ip.s6_addr32[2];
                  a.s6_addr32[3] = r->max_ip.s6_addr32[3];
                        printf("-%s", addr_to_numeric(&a));
            }
      }
      if (r->flags & IP6_NAT_RANGE_PROTO_SPECIFIED) {
            printf(":");
            printf("%hu", ntohs(r->min.tcp.port));
            if (r->max.tcp.port != r->min.tcp.port)
                  printf("-%hu", ntohs(r->max.tcp.port));
      }
}

/* Prints out the targinfo. */
static void
print(const struct ip6t_ip6 *ip,
      const struct ip6t_entry_target *target,
      int numeric)
{
      struct ip6t_natinfo *info = (void *)target;
```

```
	unsigned int i = 0;

	printf("to: ");
	for (i = 0; i < info->mr.rangesize; i++) {
		print_range(&info->mr.range[i]);
		printf(" ");
	}
}

/* Saves the union ip6t_targinfo in parsable form to stdout. */
static void
save(const struct ip6t_ip6 *ip, const struct ip6t_entry_target *target)
{
	struct ip6t_natinfo *info = (void *)target;
	unsigned int i = 0;

	for (i = 0; i < info->mr.rangesize; i++) {
		printf("--to-source ");
		print_range(&info->mr.range[i]);
		printf(" ");
	}
}

static
struct ip6tables_target snat
= { NULL,
    "SNAT",
    IPTABLES_VERSION,
    IP6T_ALIGN(sizeof(struct ip6_nat_multi_range)),
    IP6T_ALIGN(sizeof(struct ip6_nat_multi_range)),
    &help,
    &init,
    &parse,
    &final_check,
    &print,
    &save,
    opts
};

void _init(void)
{
	register_target6(&snat);
}
```

APPENDIX D. TESTING RESULTS

This appendix contains the Ethereal outputs for a series of connectivity tests. The purpose of these tests was to verify the functionality of the NAT implementation for several different protocols. The following sections will be labeled by the testing program, the machine on which the Ethereal output is collected and, if applicable, the interface of the system. The following figure shows the topology of the MYSEA IPv6 NAT testing environment. It illustrates the locations of the machines as well as their IPv6 addresses and MAC addresses.

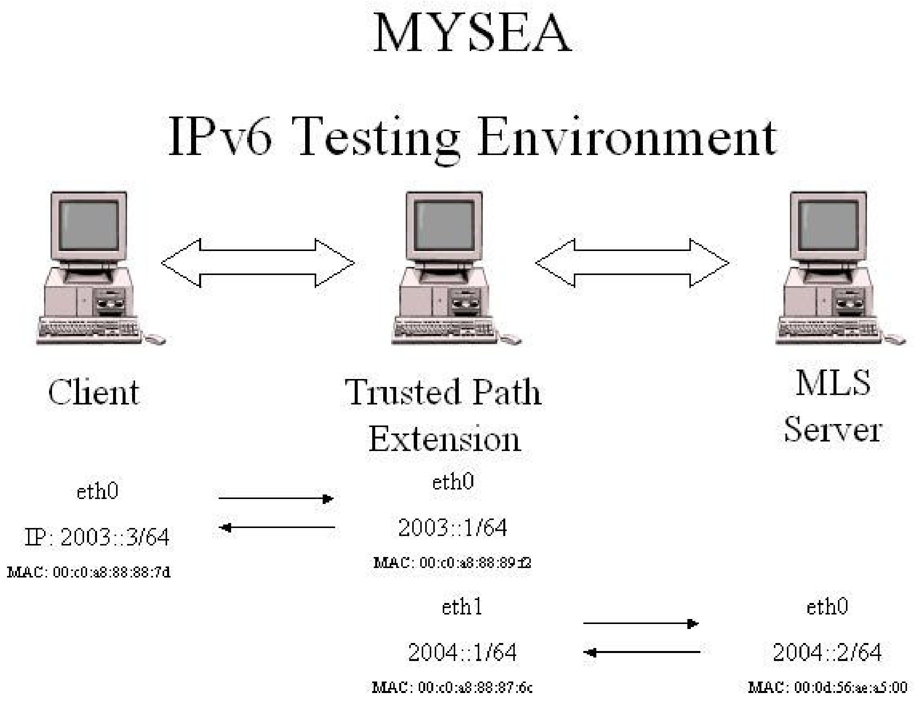


Figure 16. MYSEA IPv6 NAT Testing Environment

PING6 – CLIENT

This Ethereal output shows the packet sequence as seen by the eth0 interface of the client. The importance of this sequence is that an echo request was successfully sent to 2004::2 and the reply was received by 2003::3. The following command was issued by the client to produce this result:

```
# ping6 –c 1 2004::2
```

```
No.     Time        Source                Destination           Protocol Info
      1 0.000000    2003::3               ff02::1:ff00:1        ICMPv6   Neighbor
solicitation

Frame 1 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 33:33:ff:00:00:01
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      2 0.000213    2003::1               2003::3               ICMPv6   Neighbor
advertisement

Frame 2 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      3 0.000230    2003::3               2004::2               ICMPv6   Echo request

Frame 3 (118 bytes on wire, 118 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      4 0.001486    2004::2               2003::3               ICMPv6   Echo reply

Frame 4 (118 bytes on wire, 118 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      5 4.999079    fe80::2c0:a8ff:fe88:89f2 2003::3               ICMPv6   Neighbor
solicitation

Frame 5 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Internet Control Message Protocol v6
```

```
No.     Time        Source                Destination           Protocol Info
      6 4.999106    2003::3               fe80::2c0:a8ff:fe88:89f2 ICMPv6   Neighbor
advertisement

Frame 6 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Internet Control Message Protocol v6
```

PING6 – SERVER

This Ethereal output shows the packet sequence as seen by the eth0 interface of the server. The importance of this sequence is that an echo request was successfully forwarded to 2004::2 by the TPE at the address 2004::1. The following command was issued by the client to produce this result:

```
# ping6 –c 1 2004::2
```

```
No.     Time        Source                Destination           Protocol Info
      1 0.000000    2004::1               ff02::1:ff00:2        ICMPv6   Neighbor
solicitation

Frame 1 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 33:33:ff:00:00:02
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      2 0.000036    2004::2               2004::1               ICMPv6   Neighbor
advertisement

Frame 2 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      3 0.000167    2004::1               2004::2               ICMPv6   Echo request

Frame 3 (118 bytes on wire, 118 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      4 0.000181    2004::2               2004::1               ICMPv6   Echo reply

Frame 4 (118 bytes on wire, 118 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Internet Control Message Protocol v6
```

```
No.     Time        Source                Destination           Protocol Info
      5 4.995768    fe80::20d:56ff:feae:a500 2004::1               ICMPv6   Neighbor
solicitation

Frame 5 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      6 4.995992    2004::1               fe80::20d:56ff:feae:a500 ICMPv6   Neighbor
advertisement

Frame 6 (78 bytes on wire, 78 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      7 9.995195    fe80::2c0:a8ff:fe88:876c fe80::20d:56ff:feae:a500 ICMPv6   Neighbor
solicitation

Frame 7 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      8 9.995222    fe80::20d:56ff:feae:a500 fe80::2c0:a8ff:fe88:876c ICMPv6   Neighbor
advertisement

Frame 8 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Internet Control Message Protocol v6
```

PING6 – TPE – ETH0

This *tcpdump* output shows the packet sequence as seen by the eth0 interface of the TPE. Tcpdump was used on the TPE because it is a native OS program and would not introduce new code to the kernel. The importance of this sequence is that an echo request from 2003::3 was forwarded to 2004::2 and the resulting reply was again forwarded to 2003::3. Note that at this point, eth0 on the TPE, the address of the client is still the true address, i.e., 2003::3. The following command was issued by the client to produce this result:

```
# ping6 –c 1 2004::2
```

```
14:21:05.566665 2003::3 > ff02::1:ff00:1: icmp6: neighbor sol: who has 2003::1(src
lladdr: 00:c0:a8:88:88:7d) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2003 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0003 ff02 0000 0000 0000        ................
0x0020     0000 0001 ff00 0001 8700 07ce 0000 0000        ................
0x0030     2003 0000 0000 0000 0000 0000 0000 0001        ................
0x0040     0101 00c0 a888 887d                            .......}


14:21:05.566802 2003::1 > 2003::3: icmp6: neighbor adv: tgt is 2003::1(RSO)(tgt lladdr:
00:c0:a8:88:89:f2) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2003 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0001 2003 0000 0000 0000        ................
0x0020     0000 0000 0000 0003 8800 025a e000 0000        ...........Z....
0x0030     2003 0000 0000 0000 0000 0000 0000 0001        ................
0x0040     0201 00c0 a888 89f2                            ........


14:21:05.566893 2003::3 > 2004::2: icmp6: echo request (len 64, hlim 64)
0x0000     6000 0000 0040 3a40 2003 0000 0000 0000        `....@:@........
0x0010     0000 0000 0000 0003 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 8000 78ae ea04 0001        ..........x.....
0x0030     3817 b540 fc69 0800 0809 0a0b 0c0d 0e0f        8..@.i..........
0x0040     1011 1213 1415 1617 1819 1a1b 1c1d 1e1f        ................
0x0050     2021                                           .!


14:21:05.568097 2004::2 > 2003::3: icmp6: echo reply (len 64, hlim 63)
0x0000     6000 0000 0040 3a3f 2004 0000 0000 0000        `....@:?........
0x0010     0000 0000 0000 0002 2003 0000 0000 0000        ................
0x0020     0000 0000 0000 0003 8100 77ae ea04 0001        ..........w.....
0x0030     3817 b540 fc69 0800 0809 0a0b 0c0d 0e0f        8..@.i..........
0x0040     1011 1213 1415 1617 1819 1a1b 1c1d 1e1f        ................
0x0050     2021                                           .!


14:21:10.565796 fe80::2c0:a8ff:fe88:89f2 > 2003::3: icmp6: neighbor sol: who has
2003::3(src lladdr: 00:c0:a8:88:89:f2) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff fe80 0000 0000 0000        `.....:.........
0x0010     02c0 a8ff fe88 89f2 2003 0000 0000 0000        ................
0x0020     0000 0000 0000 0003 8700 d1a0 0000 0000        ................
0x0030     2003 0000 0000 0000 0000 0000 0000 0003        ................
0x0040     0101 00c0 a888 89f2                            ........


14:21:10.565871 2003::3 > fe80::2c0:a8ff:fe88:89f2: icmp6: neighbor adv: tgt is
2003::3(SO)(tgt lladdr: 00:c0:a8:88:88:7d) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2003 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0003 fe80 0000 0000 0000        ................
0x0020     02c0 a8ff fe88 89f2 8800 7115 6000 0000        ..........q.`...
0x0030     2003 0000 0000 0000 0000 0000 0000 0003        ................
0x0040     0201 00c0 a888 887d                            .......}
```

PING6 – TPE – ETH1

This *tcpdump* output shows the packet sequence as seen by the eth1 interface of the TPE. The importance of this sequence is that an echo request from the client received at 2004::1 was successfully forwarded to 2004::2 and the resulting reply was receieved by 2004::1. Note that the address of the client is now masked by the NAT mechanism. The following command was issued by the client to produce this result:

```
# ping6 –c 1 2004::2
```

```
14:21:05.567757 2004::1 > ff02::1:ff00:2: icmp6: neighbor sol: who has 2004::2(src
lladdr: 00:c0:a8:88:87:6c) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2004 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0001 ff02 0000 0000 0000        ................
0x0020     0000 0001 ff00 0002 8700 08dd 0000 0000        ................
0x0030     2004 0000 0000 0000 0000 0000 0000 0002        ................
0x0040     0101 00c0 a888 876c                            .......l


14:21:05.567861 2004::2 > 2004::1: icmp6: neighbor adv: tgt is 2004::2(SO)(tgt lladdr:
00:0d:56:ae:a5:00) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2004 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0002 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0001 8800 b9d6 6000 0000        ............`...
0x0030     2004 0000 0000 0000 0000 0000 0000 0002        ................
0x0040     0201 000d 56ae a500                            ....V...


14:21:05.567924 2004::1 > 2004::2: icmp6: echo request (len 64, hlim 63)
0x0000     6000 0000 0040 3a3f 2004 0000 0000 0000        `....@:?........
0x0010     0000 0000 0000 0001 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 8000 78af ea04 0001        ..........x.....
0x0030     3817 b540 fc69 0800 0809 0a0b 0c0d 0e0f        8..@.i..........
0x0040     1011 1213 1415 1617 1819 1a1b 1c1d 1e1f        ................
0x0050     2021                                           .!


14:21:05.568008 2004::2 > 2004::1: icmp6: echo reply (len 64, hlim 64)
0x0000     6000 0000 0040 3a40 2004 0000 0000 0000        `....@:@........
0x0010     0000 0000 0000 0002 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0001 8100 77af ea04 0001        ..........w.....
0x0030     3817 b540 fc69 0800 0809 0a0b 0c0d 0e0f        8..@.i..........
0x0040     1011 1213 1415 1617 1819 1a1b 1c1d 1e1f        ................
0x0050     2021                                           .!


14:21:10.563537 fe80::20d:56ff:feae:a500 > 2004::1: icmp6: neighbor sol: who has
2004::1(src lladdr: 00:0d:56:ae:a5:00) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff fe80 0000 0000 0000        `.....:.........
0x0010     020d 56ff feae a500 2004 0000 0000 0000        ..V.............
0x0020     0000 0000 0000 0001 8700 40a1 0000 0000        ..........@.....
0x0030     2004 0000 0000 0000 0000 0000 0000 0001        ................
0x0040     0101 000d 56ae a500                            ....V...


14:21:10.563690 2004::1 > fe80::20d:56ff:feae:a500: icmp6: neighbor adv: tgt is
2004::1(RS) (len 24, hlim 255)
0x0000     6000 0000 0018 3aff 2004 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0001 fe80 0000 0000 0000        ................
0x0020     020d 56ff feae a500 8800 7c65 c000 0000        ..V.......|e....
0x0030     2004 0000 0000 0000 0000 0000 0000 0001        ................


14:21:15.562827 fe80::2c0:a8ff:fe88:876c > fe80::20d:56ff:feae:a500: icmp6: neighbor sol:
who has fe80::20d:56ff:feae:a500(src lladdr: 00:c0:a8:88:87:6c) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff fe80 0000 0000 0000        `.....:.........
```

```
0x0010     02c0 a8ff fe88 876c fe80 0000 0000 0000       .......l........
0x0020     020d 56ff feae a500 8700 203f 0000 0000       ..V........?....
0x0030     fe80 0000 0000 0000 020d 56ff feae a500       ..........V.....
0x0040     0101 00c0 a888 876c                           .......l


14:21:15.562925 fe80::20d:56ff:feae:a500 > fe80::2c0:a8ff:fe88:876c: icmp6: neighbor adv:
tgt is fe80::20d:56ff:feae:a500(SO)(tgt lladdr: 00:0d:56:ae:a5:00) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff fe80 0000 0000 0000       `.....:.........
0x0010     020d 56ff feae a500 fe80 0000 0000 0000       ..V.............
0x0020     02c0 a8ff fe88 876c 8800 f337 6000 0000       .......l...7`...
0x0030     fe80 0000 0000 0000 020d 56ff feae a500       ..........V.....
0x0040     0201 000d 56ae a500                           ....V...
```

RLOGIN – CLIENT

This Ethereal output shows the packet sequence as seen by the eth0 interface of the client. The importance of this sequence is that the *rlogin* sequence was successfully sent to 2004::2 and the replies were received by 2003::3. The following command was issued by the client to produce this result:

```
# rlogin 2004::2
```

```
No.     Time        Source                Destination           Protocol Info
      1 0.000000    2003::3               2004::2               TCP      1023 > login
[SYN] Seq=0 Ack=0 Win=5760 Len=0 MSS=1440 TSV=1146738 TSER=0 WS=0

Frame 1 (94 bytes on wire, 94 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 0, Ack:
0, Len: 0



No.     Time        Source                Destination           Protocol Info
      2 0.001140    fe80::2c0:a8ff:fe88:89f2 ff02::1:ff00:3        ICMPv6   Neighbor
solicitation

Frame 2 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 33:33:ff:00:00:03
Internet Protocol Version 6
Internet Control Message Protocol v6



No.     Time        Source                Destination           Protocol Info
      3 0.001168    2003::3               fe80::2c0:a8ff:fe88:89f2 ICMPv6   Neighbor
advertisement

Frame 3 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Internet Control Message Protocol v6



No.     Time        Source                Destination           Protocol Info
```

```
      4 0.001278    2004::2               2003::3               TCP      login > 1023
[SYN, ACK] Seq=0 Ack=1 Win=5712 Len=0 MSS=1440 TSV=27902 TSER=1146738 WS=0

Frame 4 (94 bytes on wire, 94 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Transmission Control Protocol, Src Port: login (513), Dst Port: 1023 (1023), Seq: 0, Ack:
1, Len: 0



No.     Time        Source                Destination           Protocol Info
      5 0.001295    2003::3               2004::2               TCP      1023 > login
[ACK] Seq=1 Ack=1 Win=5760 Len=0 TSV=1146738 TSER=27902

Frame 5 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 1, Ack:
1, Len: 0



No.     Time        Source                Destination           Protocol Info
      6 0.001365    2003::3               2004::2               Rlogin   User name: root,
Start Handshake

Frame 6 (87 bytes on wire, 87 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 1, Ack:
1, Len: 1
Rlogin Protocol



No.     Time        Source                Destination           Protocol Info
      7 0.001535    2004::2               2003::3               TCP      login > 1023
[ACK] Seq=1 Ack=2 Win=5712 Len=0 TSV=27903 TSER=1146738

Frame 7 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Transmission Control Protocol, Src Port: login (513), Dst Port: 1023 (1023), Seq: 1, Ack:
2, Len: 0



No.     Time        Source                Destination           Protocol Info
      8 0.001658    2003::3               2004::2               Rlogin   User name: root,
Data: root

Frame 8 (109 bytes on wire, 109 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 2, Ack:
1, Len: 23
Rlogin Protocol



No.     Time        Source                Destination           Protocol Info
      9 0.001830    2004::2               2003::3               TCP      login > 1023
[ACK] Seq=1 Ack=25 Win=5712 Len=0 TSV=27903 TSER=1146738

Frame 9 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Transmission Control Protocol, Src Port: login (513), Dst Port: 1023 (1023), Seq: 1, Ack:
25, Len: 0
```

```
No.     Time        Source                Destination           Protocol Info
     10 0.154152    2004::2               2003::3               Rlogin   User name: root,
Start Handshake

Frame 10 (87 bytes on wire, 87 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Transmission Control Protocol, Src Port: login (513), Dst Port: 1023 (1023), Seq: 1, Ack:
25, Len: 1
Rlogin Protocol



No.     Time        Source                Destination           Protocol Info
     11 0.154329    2003::3               2004::2               TCP      1023 > login
[ACK] Seq=25 Ack=2 Win=5760 Len=0 TSV=1146753 TSER=27918

Frame 11 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 25,
Ack: 2, Len: 0



No.     Time        Source                Destination           Protocol Info
     12 0.155758    2004::2               2003::3               Rlogin   User name: root,
Data: Password:

Frame 12 (96 bytes on wire, 96 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Transmission Control Protocol, Src Port: login (513), Dst Port: 1023 (1023), Seq: 2, Ack:
25, Len: 10
Rlogin Protocol



No.     Time        Source                Destination           Protocol Info
     13 0.155898    2003::3               2004::2               TCP      1023 > login
[ACK] Seq=25 Ack=12 Win=5760 Len=0 TSV=1146753 TSER=27918

Frame 13 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 25,
Ack: 12, Len: 0
```

RLOGIN – SERVER

This Ethereal output shows the packet sequence as seen by the eth0 interface of the server. The importance of this sequence is that the *rlogin* request was successfully forwarded to 2004::2 by the TPE at the address 2004::1. Note that the address of the client is successfully masked by the NAT mechanism in the TPE. The following command was issued by the client to produce this result:

```
# rlogin 2004::2
```

```
No.     Time        Source                Destination           Protocol Info
      1 0.000000    fe80::20d:56ff:feae:a500 ff02::1:ff00:2        ICMPv6   Multicast
listener report

Frame 1 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 33:33:ff:00:00:02
Internet Protocol Version 6
Hop-by-hop Option Header
Internet Control Message Protocol v6



No.     Time        Source                Destination           Protocol Info
      2 61.169069   2004::1               ff02::1:ff00:2        ICMPv6   Neighbor
solicitation

Frame 2 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 33:33:ff:00:00:02
Internet Protocol Version 6
Internet Control Message Protocol v6



No.     Time        Source                Destination           Protocol Info
      3 61.169105   2004::2               2004::1               ICMPv6   Neighbor
advertisement

Frame 3 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Internet Control Message Protocol v6



No.     Time        Source                Destination           Protocol Info
      4 61.169243   2004::1               2004::2               TCP      1023 > login
[SYN] Seq=0 Ack=0 Win=5760 Len=0 MSS=1440 TSV=1146738 TSER=0 WS=0

Frame 4 (94 bytes on wire, 94 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 0, Ack:
0, Len: 0



No.     Time        Source                Destination           Protocol Info
      5 61.169504   2004::2               2004::1               TCP      login > 1023
[SYN, ACK] Seq=0 Ack=1 Win=5712 Len=0 MSS=1440 TSV=27902 TSER=1146738 WS=0

Frame 5 (94 bytes on wire, 94 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Transmission Control Protocol, Src Port: login (513), Dst Port: 1023 (1023), Seq: 0, Ack:
1, Len: 0



No.     Time        Source                Destination           Protocol Info
      6 61.170227   2004::1               2004::2               TCP      1023 > login
[ACK] Seq=1 Ack=1 Win=5760 Len=0 TSV=1146738 TSER=27902

Frame 6 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 1, Ack:
1, Len: 0
```

```
No.     Time        Source                Destination           Protocol Info
      7 61.170289   2004::1               2004::2               Rlogin   User name: root,
Start Handshake

Frame 7 (87 bytes on wire, 87 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 1, Ack:
1, Len: 1
Rlogin Protocol



No.     Time        Source                Destination           Protocol Info
      8 61.170302   2004::2               2004::1               TCP      login > 1023
[ACK] Seq=1 Ack=2 Win=5712 Len=0 TSV=27903 TSER=1146738

Frame 8 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Transmission Control Protocol, Src Port: login (513), Dst Port: 1023 (1023), Seq: 1, Ack:
2, Len: 0



No.     Time        Source                Destination           Protocol Info
      9 61.170592   2004::1               2004::2               Rlogin   User name: root,
Data: root

Frame 9 (109 bytes on wire, 109 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 2, Ack:
1, Len: 23
Rlogin Protocol



No.     Time        Source                Destination           Protocol Info
     10 61.170597   2004::2               2004::1               TCP      login > 1023
[ACK] Seq=1 Ack=25 Win=5712 Len=0 TSV=27903 TSER=1146738

Frame 10 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Transmission Control Protocol, Src Port: login (513), Dst Port: 1023 (1023), Seq: 1, Ack:
25, Len: 0



No.     Time        Source                Destination           Protocol Info
     11 61.192370   2004::2               2004::1               TCP      32803 > auth
[SYN] Seq=0 Ack=0 Win=5760 Len=0 MSS=1440 TSV=27905 TSER=0 WS=0

Frame 11 (94 bytes on wire, 94 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 32803 (32803), Dst Port: auth (113), Seq: 0,
Ack: 0, Len: 0



No.     Time        Source                Destination           Protocol Info
     12 61.192547   2004::1               2004::2               TCP      auth > 32803
[RST, ACK] Seq=0 Ack=0 Win=0 Len=0

Frame 12 (74 bytes on wire, 74 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
```

```
Transmission Control Protocol, Src Port: auth (113), Dst Port: 32803 (32803), Seq: 0,
Ack: 0, Len: 0


No.     Time        Source                Destination           Protocol Info
     13 61.322880   2004::2               2004::1               Rlogin   User name: root,
Start Handshake

Frame 13 (87 bytes on wire, 87 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Transmission Control Protocol, Src Port: login (513), Dst Port: 1023 (1023), Seq: 1, Ack:
25, Len: 1
Rlogin Protocol


No.     Time        Source                Destination           Protocol Info
     14 61.323266   2004::1               2004::2               TCP      1023 > login
[ACK] Seq=25 Ack=2 Win=5760 Len=0 TSV=1146753 TSER=27918

Frame 14 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 25,
Ack: 2, Len: 0


No.     Time        Source                Destination           Protocol Info
     15 61.324522   2004::2               2004::1               Rlogin   User name: root,
Data: Password:

Frame 15 (96 bytes on wire, 96 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Transmission Control Protocol, Src Port: login (513), Dst Port: 1023 (1023), Seq: 2, Ack:
25, Len: 10
Rlogin Protocol


No.     Time        Source                Destination           Protocol Info
     16 61.324830   2004::1               2004::2               TCP      1023 > login
[ACK] Seq=25 Ack=12 Win=5760 Len=0 TSV=1146753 TSER=27918

Frame 16 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 1023 (1023), Dst Port: login (513), Seq: 25,
Ack: 12, Len: 0
```

RLOGIN – TPE – ETH0

This *tcpdump* output shows the packet sequence as seen by the eth0 interface of the TPE. The importance of this sequence is that the *rlogin* request from 2003::3 was forwarded to 2004::2 and the resulting reply was again forwarded to 2003::3. Note that at this point, the address of the client is still the true address. The following command was issued by the client to produce this result:

```
# rlogin 2004::2
```

```
17:28:06.644285 2003::3.1023 > 2004::2.login: S [tcp sum ok] 2268744467:2268744467(0) win
5760 <mss 1440,sackOK,timestamp 1146738 0,nop,wscale 0> (len 40, hlim 64)
0x0000     6000 0000 0028 0640 2003 0000 0000 0000       `....(.@........
0x0010     0000 0000 0000 0003 2004 0000 0000 0000       ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b13       .............:K.
0x0030     0000 0000 a002 1680 99be 0000 0204 05a0       ................
0x0040     0402 080a 0011 7f72 0000 0000 0103 0300       .......r........


17:28:06.645375 fe80::2c0:a8ff:fe88:89f2 > ff02::1:ff00:3: icmp6: neighbor sol: who has
2003::3(src lladdr: 00:c0:a8:88:89:f2) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff fe80 0000 0000 0000       `.....:.........
0x0010     02c0 a8ff fe88 89f2 ff02 0000 0000 0000       ................
0x0020     0000 0001 ff00 0003 8700 f39e 0000 0000       ................
0x0030     2003 0000 0000 0000 0000 0000 0000 0003       ................
0x0040     0101 00c0 a888 89f2                            ........


17:28:06.645447 2003::3 > fe80::2c0:a8ff:fe88:89f2: icmp6: neighbor adv: tgt is
2003::3(SO)(tgt lladdr: 00:c0:a8:88:88:7d) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2003 0000 0000 0000       `.....:.........
0x0010     0000 0000 0000 0003 fe80 0000 0000 0000       ................
0x0020     02c0 a8ff fe88 89f2 8800 7115 6000 0000       ..........q.`...
0x0030     2003 0000 0000 0000 0000 0000 0000 0003       ................
0x0040     0201 00c0 a888 887d                            .......}


17:28:06.645510 2004::2.login > 2003::3.1023: S [tcp sum ok] 3072001763:3072001763(0) ack
2268744468 win 5712 <mss 1440,sackOK,timestamp 27902 1146738,nop,wscale 0> (len 40, hlim
63)
0x0000     6000 0000 0028 063f 2004 0000 0000 0000       `....(.?........
0x0010     0000 0000 0000 0002 2003 0000 0000 0000       ................
0x0020     0000 0000 0000 0003 0201 03ff b71b 06e3       ................
0x0030     873a 4b14 a012 1650 6ee0 0000 0204 05a0       .:K....Pn.......
0x0040     0402 080a 0000 6cfe 0011 7f72 0103 0300       ......l....r....


17:28:06.645574 2003::3.1023 > 2004::2.login: . [tcp sum ok] 1:1(0) ack 1 win 5760
<nop,nop,timestamp 1146738 27902> (len 32, hlim 64)
0x0000     6000 0000 0020 0640 2003 0000 0000 0000       `......@........
0x0010     0000 0000 0000 0003 2004 0000 0000 0000       ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b14       .............:K.
0x0030     b71b 06e4 8010 1680 9d61 0000 0101 080a       .........a......
0x0040     0011 7f72 0000 6cfe                            ...r..l.


17:28:06.645644 2003::3.1023 > 2004::2.login: P [tcp sum ok] 1:2(1) ack 1 win 5760
<nop,nop,timestamp 1146738 27902> (len 33, hlim 64)
0x0000     6000 0000 0021 0640 2003 0000 0000 0000       `....!.@........
0x0010     0000 0000 0000 0003 2004 0000 0000 0000       ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b14       .............:K.
0x0030     b71b 06e4 8018 1680 9d58 0000 0101 080a       .........X......
0x0040     0011 7f72 0000 6cfe 00                         ...r..l..


17:28:06.645769 2004::2.login > 2003::3.1023: . [tcp sum ok] 1:1(0) ack 2 win
5712 <nop,nop,timestamp 27903 1146738> (len 32, hlim 63)
0x0000     6000 0000 0020 063f 2004 0000 0000 0000       `......?........
0x0010     0000 0000 0000 0002 2003 0000 0000 0000       ................
0x0020     0000 0000 0000 0003 0201 03ff b71b 06e4       ................
0x0030     873a 4b15 8010 1650 9d8f 0000 0101 080a       .:K....P........
0x0040     0000 6cff 0011 7f72                            ..l....r


17:28:06.645940 2003::3.1023 > 2004::2.login: P 2:25(23) ack 1 win 5760
<nop,nop,timestamp 1146738 27903> (len 55, hlim 64)
0x0000     6000 0000 0037 0640 2003 0000 0000 0000        `....7.@........
0x0010     0000 0000 0000 0003 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b15        .............:K.
0x0030     b71b 06e4 8018 1680 6e3d 0000 0101 080a        ........n=......
0x0040     0011 7f72 0000 6cff 726f 6f74 0061 646d        ...r..l.root.adm
0x0050     696e                                           in


17:28:06.646065 2004::2.login > 2003::3.1023: . [tcp sum ok] 1:1(0) ack 25 win 5712
<nop,nop,timestamp 27903 1146738> (len 32, hlim 63)
0x0000     6000 0000 0020 063f 2004 0000 0000 0000        `......?........
0x0010     0000 0000 0000 0002 2003 0000 0000 0000        ................
0x0020     0000 0000 0000 0003 0201 03ff b71b 06e4        ................
0x0030     873a 4b2c 8010 1650 9d78 0000 0101 080a        .:K,...P.x......
0x0040     0000 6cff 0011 7f72                            ..l....r


17:28:06.798386 2004::2.login > 2003::3.1023: P [tcp sum ok] 1:2(1) ack 25 win 5712
<nop,nop,timestamp 27918 1146738> (len 33, hlim 63)
0x0000     6000 0000 0021 063f 2004 0000 0000 0000        `....!.?........
0x0010     0000 0000 0000 0002 2003 0000 0000 0000        ................
0x0020     0000 0000 0000 0003 0201 03ff b71b 06e4        ................
0x0030     873a 4b2c 8018 1650 9d60 0000 0101 080a        .:K,...P.`......
0x0040     0000 6d0e 0011 7f72 00                         ..m....r.


17:28:06.798613 2003::3.1023 > 2004::2.login: . [tcp sum ok] 25:25(0) ack 2 win 5760
<nop,nop,timestamp 1146753 27918> (len 32, hlim 64)
0x0000     6000 0000 0020 0640 2003 0000 0000 0000        `......@........
0x0010     0000 0000 0000 0003 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b2c        .............:K,
0x0030     b71b 06e5 8010 1680 9d29 0000 0101 080a        .........)......
0x0040     0011 7f81 0000 6d0e                            ......m.


17:28:06.799994 2004::2.login > 2003::3.1023: P [tcp sum ok] 2:12(10) ack 25 win 5712
<nop,nop,timestamp 27918 1146753> (len 42, hlim 63)
0x0000     6000 0000 002a 063f 2004 0000 0000 0000        `....*.?........
0x0010     0000 0000 0000 0002 2003 0000 0000 0000        ................
0x0020     0000 0000 0000 0003 0201 03ff b71b 06e5        ................
0x0030     873a 4b2c 8018 1650 b57e 0000 0101 080a        .:K,...P.~......
0x0040     0000 6d0e 0011 7f81 5061 7373 776f 7264        ..m.....Password
0x0050     3a20                                           :.


17:28:06.800181 2003::3.1023 > 2004::2.login: . [tcp sum ok] 25:25(0) ack 12 win 5760
<nop,nop,timestamp 1146753 27918> (len 32, hlim 64)
0x0000     6000 0000 0020 0640 2003 0000 0000 0000        `......@........
0x0010     0000 0000 0000 0003 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b2c        .............:K,
0x0030     b71b 06ef 8010 1680 9d1f 0000 0101 080a        ................
0x0040     0011 7f81 0000 6d0e                            ......m.
```

RLOGIN – TPE – ETH1

This *tcpdump* output shows the packet sequence as seen by the eth1 interface of the TPE. The importance of this sequence is that an *rlogin* session from the client was translated to appear as if it came from 2004::1. The translated packet was successfully communicated to 2004::2 and the resulting replies were received by 2004::1. Note that the address of the client is now masked by the NAT mechanism. The following command was issued by the client to produce this result:

```
     # rlogin 2004::2
```

```
17:28:06.644443 2004::1 > ff02::1:ff00:2: icmp6: neighbor sol: who has 2004::2(src
lladdr: 00:c0:a8:88:87:6c) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2004 0000 0000 0000       `.....:.........
0x0010     0000 0000 0000 0001 ff02 0000 0000 0000       ................
0x0020     0000 0001 ff00 0002 8700 08dd 0000 0000       ................
0x0030     2004 0000 0000 0000 0000 0000 0000 0002       ................
0x0040     0101 00c0 a888 876c                           .......l


17:28:06.644550 2004::2 > 2004::1: icmp6: neighbor adv: tgt is 2004::2(SO)(tgt lladdr:
00:0d:56:ae:a5:00) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2004 0000 0000 0000       `.....:.........
0x0010     0000 0000 0000 0002 2004 0000 0000 0000       ................
0x0020     0000 0000 0000 0001 8800 b9d6 6000 0000       ............`...
0x0030     2004 0000 0000 0000 0000 0000 0000 0002       ................
0x0040     0201 000d 56ae a500                           ....V...


17:28:06.644623 2004::1.1023 > 2004::2.login: S [tcp sum ok] 2268744467:2268744467(0) win
5760 <mss 1440,sackOK,timestamp 1146738 0,nop,wscale 0> (len 40, hlim 63)
0x0000     6000 0000 0028 063f 2004 0000 0000 0000       `....(.?........
0x0010     0000 0000 0000 0001 2004 0000 0000 0000       ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b13       .............:K.
0x0030     0000 0000 a002 1680 99bf 0000 0204 05a0       ................
0x0040     0402 080a 0011 7f72 0000 0000 0103 0300       .......r........


17:28:06.644947 2004::2.login > 2004::1.1023: S [tcp sum ok] 3072001763:3072001763(0) ack
2268744468 win 5712 <mss 1440,sackOK,timestamp 27902 1146738,nop,wscale 0> (len 40, hlim
64)
0x0000     6000 0000 0028 0640 2004 0000 0000 0000       `....(.@........
0x0010     0000 0000 0000 0002 2004 0000 0000 0000       ................
0x0020     0000 0000 0000 0001 0201 03ff b71b 06e3       ................
0x0030     873a 4b14 a012 1650 6ee1 0000 0204 05a0       .:K....Pn.......
0x0040     0402 080a 0000 6cfe 0011 7f72 0103 0300       ......l....r....


17:28:06.645599 2004::1.1023 > 2004::2.login: . [tcp sum ok] 1:1(0) ack 1 win 5760
<nop,nop,timestamp 1146738 27902> (len 32, hlim 63)
0x0000     6000 0000 0020 063f 2004 0000 0000 0000       `......?........
0x0010     0000 0000 0000 0001 2004 0000 0000 0000       ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b14       .............:K.
0x0030     b71b 06e4 8010 1680 9d62 0000 0101 080a       .........b......
0x0040     0011 7f72 0000 6cfe                           ...r..l.


17:28:06.645667 2004::1.1023 > 2004::2.login: P [tcp sum ok] 1:2(1) ack 1 win 5760
<nop,nop,timestamp 1146738 27902> (len 33, hlim 63)
0x0000     6000 0000 0021 063f 2004 0000 0000 0000       `....!.?........
0x0010     0000 0000 0000 0001 2004 0000 0000 0000       ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b14       .............:K.
0x0030     b71b 06e4 8018 1680 9d59 0000 0101 080a       .........Y......
0x0040     0011 7f72 0000 6cfe 00                        ...r..l..


17:28:06.645745 2004::2.login > 2004::1.1023: . [tcp sum ok] 1:1(0) ack 2 win 5712
<nop,nop,timestamp 27903 1146738> (len 32, hlim 64)
0x0000     6000 0000 0020 0640 2004 0000 0000 0000       `......@........
0x0010     0000 0000 0000 0002 2004 0000 0000 0000       ................
0x0020     0000 0000 0000 0001 0201 03ff b71b 06e4       ................
0x0030     873a 4b15 8010 1650 9d90 0000 0101 080a       .:K....P........
0x0040     0000 6cff 0011 7f72                           ..l....r
```

```
17:28:06.645966 2004::1.1023 > 2004::2.login: P 2:25(23) ack 1 win 5760
<nop,nop,timestamp 1146738 27903> (len 55, hlim 63)
0x0000     6000 0000 0037 063f 2004 0000 0000 0000        `....7.?........
0x0010     0000 0000 0000 0001 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b15        .............:K.
0x0030     b71b 06e4 8018 1680 6e3e 0000 0101 080a        ........n>......
0x0040     0011 7f72 0000 6cff 726f 6f74 0061 646d        ...r..l.root.adm
0x0050     696e                                           in


17:28:06.646040 2004::2.login > 2004::1.1023: . [tcp sum ok] 1:1(0) ack 25 win 5712
<nop,nop,timestamp 27903 1146738> (len 32, hlim 64)
0x0000     6000 0000 0020 0640 2004 0000 0000 0000        `......@........
0x0010     0000 0000 0000 0002 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0001 0201 03ff b71b 06e4        ................
0x0030     873a 4b2c 8010 1650 9d79 0000 0101 080a        .:K,...P.y......
0x0040     0000 6cff 0011 7f72                            ..l....r


17:28:06.667824 2004::2.32803 > 2004::1.auth: S [tcp sum ok] 3060071179:3060071179(0) win
5760 <mss 1440,sackOK,timestamp 27905 0,nop,wscale 0> (len 40, hlim 64)
0x0000     6000 0000 0028 0640 2004 0000 0000 0000        `....(.@........
0x0010     0000 0000 0000 0002 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0001 8023 0071 b664 fb0b        .........#.q.d..
0x0030     0000 0000 a002 1680 528a 0000 0204 05a0        ........R.......
0x0040     0402 080a 0000 6d01 0000 0000 0103 0300        ......m.........


17:28:06.667925 2004::1.auth > 2004::2.32803: R [tcp sum ok] 0:0(0) ack 3060071180 win 0
(len 20, hlim 64)
0x0000     6000 0000 0014 0640 2004 0000 0000 0000        `......@........
0x0010     0000 0000 0000 0001 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 0071 8023 0000 0000        .........q.#....
0x0030     b664 fb0c 5014 0000 3dc0 0000                  .d..P...=...


17:28:06.798329 2004::2.login > 2004::1.1023: P [tcp sum ok] 1:2(1) ack 25 win 5712
<nop,nop,timestamp 27918 1146738> (len 33, hlim 64)
0x0000     6000 0000 0021 0640 2004 0000 0000 0000        `....!.@........
0x0010     0000 0000 0000 0002 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0001 0201 03ff b71b 06e4        ................
0x0030     873a 4b2c 8018 1650 9d61 0000 0101 080a        .:K,...P.a......
0x0040     0000 6d0e 0011 7f72 00                         ..m....r.


17:28:06.798639 2004::1.1023 > 2004::2.login: . [tcp sum ok] 25:25(0) ack 2 win 5760
<nop,nop,timestamp 1146753 27918> (len 32, hlim 63)
0x0000     6000 0000 0020 063f 2004 0000 0000 0000        `......?........
0x0010     0000 0000 0000 0001 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b2c        .............:K,
0x0030     b71b 06e5 8010 1680 9d2a 0000 0101 080a        .........*......
0x0040     0011 7f81 0000 6d0e                            ......m.


17:28:06.799969 2004::2.login > 2004::1.1023: P [tcp sum ok] 2:12(10) ack 25 win 5712
<nop,nop,timestamp 27918 1146753> (len 42, hlim 64)
0x0000     6000 0000 002a 0640 2004 0000 0000 0000        `....*.@........
0x0010     0000 0000 0000 0002 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0001 0201 03ff b71b 06e5        ................
0x0030     873a 4b2c 8018 1650 b57f 0000 0101 080a        .:K,...P........
0x0040     0000 6d0e 0011 7f81 5061 7373 776f 7264        ..m.....Password
0x0050     3a20                                           :.


17:28:06.800205 2004::1.1023 > 2004::2.login: . [tcp sum ok] 25:25(0) ack 12 win 5760
<nop,nop,timestamp 1146753 27918> (len 32, hlim 63)
0x0000     6000 0000 0020 063f 2004 0000 0000 0000        `......?........
0x0010     0000 0000 0000 0001 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 03ff 0201 873a 4b2c        .............:K,
0x0030     b71b 06ef 8010 1680 9d20 0000 0101 080a        ................
0x0040     0011 7f81 0000 6d0e                            ......m.
```

TRACEROUTE6 – CLIENT

This Ethereal output shows the packet sequence as seen by the eth0 interface of the client. The importance of this sequence is that a *traceroute6* UDP packet sequence was successfully sent to 2004::2 and the reply was received by 2003::3. The following command was issued by the client to produce this result:

```
# traceroute6 2004::2
```

```
No.      Time         Source                 Destination
Protocol Info
      1 0.000000    2003::3               2004::2               UDP      Source port:
32769  Destination port: 33434

Frame 1 (78 bytes on wire, 78 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
User Datagram Protocol, Src Port: 32769 (32769), Dst Port: 33434 (33434)
Data (16 bytes)

0000  00 00 07 b3 00 00 00 01 b1 2f b5 40 80 9f 01 00   ........./.@....


No.     Time        Source                Destination           Protocol Info
      2 0.000565    2003::1               ff02::1:ff00:3        ICMPv6   Neighbor
solicitation

Frame 2 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 33:33:ff:00:00:03
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      3 0.000602    2003::3               2003::1               ICMPv6   Neighbor
advertisement

Frame 3 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      4 0.000714    2003::1               2003::3               ICMPv6   Time exceeded
(In-transit)

Frame 4 (126 bytes on wire, 126 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Internet Control Message Protocol v6
```

```
No.     Time        Source                Destination           Protocol Info
      5 0.031311    2003::3               2004::2               UDP      Source port:
32769  Destination port: 33434

Frame 5 (78 bytes on wire, 78 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
User Datagram Protocol, Src Port: 32769 (32769), Dst Port: 33434 (33434)
Data (16 bytes)

0000  00 00 07 b3 00 00 00 02 b1 2f b5 40 d0 19 02 00   ........./.@....



No.     Time        Source                Destination           Protocol Info
      6 0.031928    2004::2               2003::3               ICMPv6   Unreachable
(Port unreachable)

Frame 6 (126 bytes on wire, 126 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Internet Control Message Protocol v6



No.     Time        Source                Destination           Protocol Info
      7 5.030159    fe80::2c0:a8ff:fe88:887d 2003::1               ICMPv6   Neighbor
solicitation

Frame 7 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Internet Control Message Protocol v6



No.     Time        Source                Destination           Protocol Info
      8 5.030330    2003::1               fe80::2c0:a8ff:fe88:887d ICMPv6   Neighbor
advertisement

Frame 8 (78 bytes on wire, 78 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Internet Control Message Protocol v6



No.     Time        Source                Destination           Protocol Info
      9 10.030400   fe80::2c0:a8ff:fe88:89f2 fe80::2c0:a8ff:fe88:887d ICMPv6   Neighbor
solicitation

Frame 9 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:89:f2, Dst: 00:c0:a8:88:88:7d
Internet Protocol Version 6
Internet Control Message Protocol v6



No.     Time        Source                Destination           Protocol Info
     10 10.030427   fe80::2c0:a8ff:fe88:887d fe80::2c0:a8ff:fe88:89f2 ICMPv6   Neighbor
advertisement

Frame 10 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:88:7d, Dst: 00:c0:a8:88:89:f2
Internet Protocol Version 6
Internet Control Message Protocol v6
```

TRACEROUTE6 – SERVER

This Ethereal output shows the packet sequence as seen by the eth0 interface of the server. The importance of this sequence is that a *traceroute6* UDP packet sequence was successfully forwarded to 2004::2 by the TPE at the address 2004::1. The following command was issued by the client to produce this result:

```
# traceroute6 2004::2
```

```
No.     Time        Source                Destination           Protocol Info
      1 0.000000    2004::1               ff02::1:ff00:2        ICMPv6   Neighbor
solicitation

Frame 1 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 33:33:ff:00:00:02
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      2 0.000035    2004::2               2004::1               ICMPv6   Neighbor
advertisement

Frame 2 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      3 0.000166    2004::1               2004::2               UDP      Source port:
32769  Destination port: 33434

Frame 3 (78 bytes on wire, 78 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
User Datagram Protocol, Src Port: 32769 (32769), Dst Port: 33434 (33434)
Data (16 bytes)

0000  00 00 07 b3 00 00 00 02 b1 2f b5 40 d0 19 02 00   ........./.@....


No.     Time        Source                Destination           Protocol Info
      4 0.000179    2004::2               2004::1               ICMPv6   Unreachable
(Port unreachable)

Frame 4 (126 bytes on wire, 126 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      5 4.994547    fe80::20d:56ff:feae:a500 2004::1               ICMPv6   Neighbor
solicitation
```

```
Frame 5 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      6 4.994766    2004::1               fe80::20d:56ff:feae:a500 ICMPv6   Neighbor
advertisement

Frame 6 (78 bytes on wire, 78 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      7 9.994194    fe80::2c0:a8ff:fe88:876c fe80::20d:56ff:feae:a500 ICMPv6   Neighbor
solicitation

Frame 7 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:c0:a8:88:87:6c, Dst: 00:0d:56:ae:a5:00
Internet Protocol Version 6
Internet Control Message Protocol v6


No.     Time        Source                Destination           Protocol Info
      8 9.994220    fe80::20d:56ff:feae:a500 fe80::2c0:a8ff:fe88:876c ICMPv6   Neighbor
advertisement

Frame 8 (86 bytes on wire, 86 bytes captured)
Ethernet II, Src: 00:0d:56:ae:a5:00, Dst: 00:c0:a8:88:87:6c
Internet Protocol Version 6
Internet Control Message Protocol v6
```

TRACEROUTE6 – TPE – ETH0

This *tcpdump* output shows the packet sequence as seen by the eth0 interface of the TPE. The importance of this sequence is that the *traceroute6* UDP packets from 2003::3 were forwarded to 2004::2 and the resulting replies were again forwarded to 2003::3. Note that at this point, the address of the client is still the true address. The following command was issued by the client to produce this result:

```
# traceroute6 2004::2
```

```
16:05:30.255708 2003::3.32769 > 2004::2.traceroute: [udp sum ok] udp 16 [hlim 1] (len 24)
0x0000     6000 0000 0018 1101 2003 0000 0000 0000        `...............
0x0010     0000 0000 0000 0003 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 8001 829a 0018 cd52        ...............R
0x0030     0000 07b3 0000 0001 b12f b540 809f 0100        ........./.@....
```

```
16:05:30.256221 2003::1 > ff02::1:ff00:3: icmp6: neighbor sol: who has 2003::3(src
lladdr: 00:c0:a8:88:89:f2) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2003 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0001 ff02 0000 0000 0000        ................
0x0020     0000 0001 ff00 0003 8700 0657 0000 0000        ...........W....
0x0030     2003 0000 0000 0000 0000 0000 0000 0003        ................
0x0040     0101 00c0 a888 89f2                            ........
```

```
16:05:30.256305 2003::3 > 2003::1: icmp6: neighbor adv: tgt is 2003::3(SO)(tgt lladdr:
00:c0:a8:88:88:7d) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2003 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0003 2003 0000 0000 0000        ................
0x0020     0000 0000 0000 0001 8800 83cd 6000 0000        ............`...
0x0030     2003 0000 0000 0000 0000 0000 0000 0003        ................
0x0040     0201 00c0 a888 887d                            .......}
```

```
16:05:30.256371 2003::1 > 2003::3: [|icmp6] (len 72, hlim 64)
0x0000     6000 0000 0048 3a40 2003 0000 0000 0000        `....H:@........
0x0010     0000 0000 0000 0001 2003 0000 0000 0000        ................
0x0020     0000 0000 0000 0003 0300 4b83 0000 0000        ..........K.....
0x0030     6000 0000 0018 1101 2003 0000 0000 0000        `...............
0x0040     0000 0000 0000 0003 2004 0000 0000 0000        ................
0x0050     0000                                           ..
```

```
16:05:30.287017 2003::3.32769 > 2004::2.traceroute: [udp sum ok] udp 16 (len 24, hlim 2)
0x0000     6000 0000 0018 1102 2003 0000 0000 0000        `...............
0x0010     0000 0000 0000 0003 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 8001 829a 0018 7cd7        ..............|.
0x0030     0000 07b3 0000 0002 b12f b540 d019 0200        ........./.@....
```

```
16:05:30.287582 2004::2 > 2003::3: [|icmp6] (len 72, hlim 63)
0x0000     6000 0000 0048 3a3f 2004 0000 0000 0000        `....H:?........
0x0010     0000 0000 0000 0002 2003 0000 0000 0000        ................
0x0020     0000 0000 0000 0003 0104 7f9e 0000 0000        ................
0x0030     6000 0000 0018 1101 2003 0000 0000 0000        `...............
0x0040     0000 0000 0000 0003 2004 0000 0000 0000        ................
0x0050     0000                                           ..
```

```
16:05:35.285973 fe80::2c0:a8ff:fe88:887d > 2003::1: icmp6: neighbor sol: who has
2003::1(src lladdr: 00:c0:a8:88:88:7d) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff fe80 0000 0000 0000        `.....:.........
0x0010     02c0 a8ff fe88 887d 2003 0000 0000 0000        .......}........
0x0020     0000 0000 0000 0001 8700 d48e 0000 0000        ................
0x0030     2003 0000 0000 0000 0000 0000 0000 0001        ................
0x0040     0101 00c0 a888 887d                            .......}
```

```
16:05:35.286097 2003::1 > fe80::2c0:a8ff:fe88:887d: icmp6: neighbor adv: tgt is
2003::1(RS) (len 24, hlim 255)
0x0000     6000 0000 0018 3aff 2003 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0001 fe80 0000 0000 0000        ................
0x0020     02c0 a8ff fe88 887d 8800 465d c000 0000        .......}..F]....
0x0030     2003 0000 0000 0000 0000 0000 0000 0001        ................
```

```
16:05:40.286260 fe80::2c0:a8ff:fe88:89f2 > fe80::2c0:a8ff:fe88:887d: icmp6: neighbor sol:
who has fe80::2c0:a8ff:fe88:887d(src lladdr: 00:c0:a8:88:89:f2) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff fe80 0000 0000 0000        `.....:.........
0x0010     02c0 a8ff fe88 89f2 fe80 0000 0000 0000        ................
0x0020     02c0 a8ff fe88 887d 8700 af1e 0000 0000        .......}........
0x0030     fe80 0000 0000 0000 02c0 a8ff fe88 887d        ...............}
0x0040     0101 00c0 a888 89f2                            ........
```

```
16:05:40.286344 fe80::2c0:a8ff:fe88:887d > fe80::2c0:a8ff:fe88:89f2: icmp6: neighbor adv:
tgt is fe80::2c0:a8ff:fe88:887d(SO)(tgt lladdr: 00:c0:a8:88:88:7d) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff fe80 0000 0000 0000        `.....:.........
0x0010     02c0 a8ff fe88 887d fe80 0000 0000 0000        .......}........
0x0020     02c0 a8ff fe88 89f2 8800 4e93 6000 0000        ..........N.`...
0x0030     fe80 0000 0000 0000 02c0 a8ff fe88 887d        ...............}
0x0040     0201 00c0 a888 887d                            .......}
```

TRACEROUTE6 – TPE – ETH1

This *tcpdump* output shows the packet sequence as seen by the eth1 interface of the TPE. The importance of this sequence is that a *traceroute6* UDP sequence from the client was translated to appear as if it came from 2004::1. The translated packet was successfully communicated to 2004::2 and the resulting replies were received by 2004::1. Note that the address of the client is now masked by the NAT mechanism. The following command was issued by the client to produce this result:

```
# traceroute6 2004::2
```

```
16:05:30.287211 2004::1 > ff02::1:ff00:2: icmp6: neighbor sol: who has 2004::2(src
lladdr: 00:c0:a8:88:87:6c) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2004 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0001 ff02 0000 0000 0000        ................
0x0020     0000 0001 ff00 0002 8700 08dd 0000 0000        ................
0x0030     2004 0000 0000 0000 0000 0000 0000 0002        ................
0x0040     0101 00c0 a888 876c                            .......l


16:05:30.287315 2004::2 > 2004::1: icmp6: neighbor adv: tgt is 2004::2(SO)(tgt lladdr:
00:0d:56:ae:a5:00) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff 2004 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0002 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0001 8800 b9d6 6000 0000        ............`...
0x0030     2004 0000 0000 0000 0000 0000 0000 0002        ................
0x0040     0201 000d 56ae a500                            ....V...


16:05:30.287382 2004::1.32769 > 2004::2.traceroute: [udp sum ok] udp 16 [hlim 1] (len 24)
0x0000     6000 0000 0018 1101 2004 0000 0000 0000        `...............
0x0010     0000 0000 0000 0001 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0002 8001 829a 0018 7cd8        ..............|.
0x0030     0000 07b3 0000 0002 b12f b540 d019 0200        ........./.@....


16:05:30.287463 2004::2 > 2004::1: [|icmp6] (len 72, hlim 64)
0x0000     6000 0000 0048 3a40 2004 0000 0000 0000        `....H:@........
0x0010     0000 0000 0000 0002 2004 0000 0000 0000        ................
0x0020     0000 0000 0000 0001 0104 4d7e 0000 0000        ..........M~....
0x0030     6000 0000 0018 1101 2004 0000 0000 0000        `...............
0x0040     0000 0000 0000 0001 2004 0000 0000 0000        ................
0x0050     0000                                           ..
```

```
16:05:35.281776 fe80::20d:56ff:feae:a500 > 2004::1: icmp6: neighbor sol: who has
2004::1(src lladdr: 00:0d:56:ae:a5:00) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff fe80 0000 0000 0000        `.....:.........
0x0010     020d 56ff feae a500 2004 0000 0000 0000        ..V.............
0x0020     0000 0000 0000 0001 8700 40a1 0000 0000        ..........@.....
0x0030     2004 0000 0000 0000 0000 0000 0000 0001        ................
0x0040     0101 000d 56ae a500                            ....V...


16:05:35.281924 2004::1 > fe80::20d:56ff:feae:a500: icmp6: neighbor adv: tgt is
2004::1(RS) (len 24, hlim 255)
0x0000     6000 0000 0018 3aff 2004 0000 0000 0000        `.....:.........
0x0010     0000 0000 0000 0001 fe80 0000 0000 0000        ................
0x0020     020d 56ff feae a500 8800 7c65 c000 0000        ..V.......|e....
0x0030     2004 0000 0000 0000 0000 0000 0000 0001        ................


16:05:40.281290 fe80::2c0:a8ff:fe88:876c > fe80::20d:56ff:feae:a500: icmp6: neighbor sol:
who has fe80::20d:56ff:feae:a500(src lladdr: 00:c0:a8:88:87:6c) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff fe80 0000 0000 0000        `.....:.........
0x0010     02c0 a8ff fe88 876c fe80 0000 0000 0000        .......l........
0x0020     020d 56ff feae a500 8700 203f 0000 0000        ..V........?....
0x0030     fe80 0000 0000 0000 020d 56ff feae a500        ..........V.....
0x0040     0101 00c0 a888 876c                            .......l


16:05:40.281387 fe80::20d:56ff:feae:a500 > fe80::2c0:a8ff:fe88:876c: icmp6: neighbor adv:
tgt is fe80::20d:56ff:feae:a500(SO)(tgt lladdr: 00:0d:56:ae:a5:00) (len 32, hlim 255)
0x0000     6000 0000 0020 3aff fe80 0000 0000 0000        `.....:.........
0x0010     020d 56ff feae a500 fe80 0000 0000 0000        ..V.............
0x0020     02c0 a8ff fe88 876c 8800 f337 6000 0000        .......l...7`...
0x0030     fe80 0000 0000 0000 020d 56ff feae a500        ..........V.....
0x0040     0201 000d 56ae a500                            ....V...
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E. USER MANUAL

This appendix contains the man page for *ip6tables* that has been modified to include use instructions for this NAT development for IPv6. The NAT description is based on the original NAT description in the man page for *iptables* and is highlighted with preceeding "***" below.

NAME
ip6tables - IPv6 packet filter administration and NAT.

SYNOPSIS
ip6tables [-t table] -[AD] chain rule-specification [options]
ip6tables [-t table] -I chain [rulenum] rule-specification [options]
ip6tables [-t table] -R chain rulenum rule-specification [options]
ip6tables [-t table] -D chain rulenum [options]
ip6tables [-t table] -[LFZ] [chain] [options]
ip6tables [-t table] -N chain
ip6tables [-t table] -X [chain]
ip6tables [-t table] -P chain target [options]
ip6tables [-t table] -E old-chain-name new-chain-name

DESCRIPTION
Ip6tables is used to set up, maintain, and inspect the tables of IPv6 packet filter rules in the Linux kernel. Several different tables may be defined. Each table contains a number of built-in chains and may also contain user-defined chains.
Each chain is a list of rules which can match a set of packets. Each rule specifies what to do with a packet that matches. This is called a `target', which may be a jump to a user-defined chain in the same table.

TARGETS
A firewall rule specifies criteria for a packet, and a target. If the packet does not match, the next rule in the chain is the examined; if it does match, then the next rule is specified by the value of the target, which can be the name of a user-defined chain or one of the special values *ACCEPT*, *DROP*, *QUEUE*, or *RETURN*.
ACCEPT means to let the packet through. *DROP* means to drop the packet on the floor. *QUEUE* means to pass the packet to userspace (if supported by the kernel). *RETURN* means stop traversing this chain and resume at the next rule in the previous (calling) chain. If the end of a built-in chain is reached or a rule in a built-in chain with target *RETURN* is matched, the target specified by the chain policy determines the fate of the packet.

TABLES
There are currently two independent tables (which tables are present at any time depends on the kernel configuration options and which modules are present), as nat table has not been implemented yet.

-t, --table *table*
This option specifies the packet matching table which the command should operate on. If the kernel is configured with automatic module loading, an attempt will be made to load the appropriate module for that table if it is not already there.
The tables are as follows:

filter:
This is the default table (if no -t option is passed). It contains the built-in chains **INPUT** (for packets coming into the box itself), **FORWARD** (for packets being routed through the box), and **OUTPUT** (for locally-generated packets).

***** nat**:
This table is consulted when a packet that creates a new connection is encountered. It consists of three built-ins: **PREROUTING** (for altering packets as soon as they come in), **OUTPUT** (for altering locally-generated packets before routing), and **POSTROUTING** (for altering packets as they are about to go out.

mangle:
This table is used for specialized packet alteration. Until kernel 2.4.17 it had two built-in chains: **PREROUTING** (for altering incoming packets before routing) and **OUTPUT** (for altering locally-generated packets before routing). Since kernel 2.4.18, three other built-in chains are also supported: **INPUT** (for packets coming into the box itself), **FORWARD** (for altering packets being routed through the box), and **POSTROUTING** (for altering packets as they are about to go out).

OPTIONS
The options that are recognized by **ip6tables** can be divided into several different groups.

COMMANDS
These options specify the specific action to perform. Only one of them can be specified on the command line unless otherwise specified below. For all the long versions of the command and option names, you need to use only enough letters to ensure that **ip6tables** can differentiate it from all other options.

-A, --append *chain rule-specification*
Append one or more rules to the end of the selected chain. When the source and/or destination names resolve to more than one address, a rule will be added for each possible address combination.

-D, --delete *chain rule-specification*
-D, --delete *chain rulenum*
Delete one or more rules from the selected chain. There are two versions of this command: the rule can be specified as a number in the chain (starting at 1 for the first rule) or a rule to match.

-I, --insert
Insert one or more rules in the selected chain as the given rule number. So, if the rule number is 1, the rule or rules are inserted at the head of the chain. This is also the default if no rule number is specified.

-R, **--replace** *chain rulenum rule-specification*
Replace a rule in the selected chain. If the source and/or destination names resolve to multiple addresses, the command will fail. Rules are numbered starting at 1.

-L, **--list** [*chain*]
List all rules in the selected chain. If no chain is selected, all chains are listed. As every other iptables command, it applies to the specified table (filter is the default), so mangle rules get listed by

```
 ip6tables -t mangle -n -L
```

Please note that it is often used with the **-n** option, in order to avoid long reverse DNS lookups. It is legal to specify the **-Z** (zero) option as well, in which case the chain(s) will be atomically listed and zeroed. The exact output is affected by the other arguments given. The exact rules are suppressed until you use

```
 ip6tables -L -v
```

-F, **--flush** [*chain*]
Flush the selected chain (all the chains in the table if none is given). This is equivalent to deleting all the rules one by one.

-Z, **--zero** [*chain*]
Zero the packet and byte counters in all chains. It is legal to specify the **-L, --list** (list) option as well, to see the counters immediately before they are cleared. (See above.)

-N, **--new-chain** *chain*
Create a new user-defined chain by the given name. There must be no target of that name already.

-X, **--delete-chain** [*chain*]
Delete the optional user-defined chain specified. There must be no references to the chain. If there are, you must delete or replace the referring rules before the chain can be deleted. If no argument is given, it will attempt to delete every non-builtin chain in the table.

-P, **--policy** *chain target*
Set the policy for the chain to the given target. See the section **TARGETS** for the legal targets. Only built-in (non-user-defined) chains can have policies, and neither built-in nor user-defined chains can be policy targets.

-E, **--rename-chain** *old-chain new-chain*
Rename the user specified chain to the user supplied name. This is cosmetic, and has no effect on the structure of the table.
-h
Help. Give a (currently very brief) description of the command syntax.

PARAMETERS
The following parameters make up a rule specification (as used in the add, delete, insert, replace and append commands).

-p, **--protocol** [!] *protocol*
The protocol of the rule or of the packet to check. The specified protocol can be one of *tcp*, *udp*, *ipv6-icmp|icmpv6*, or *all*, or it can be a numeric value, representing one of these protocols or a different one. A protocol name from /etc/protocols is also allowed. A "!"

argument before the protocol inverts the test. The number zero is equivalent to *all*. Protocol *all* will match with all protocols and is taken as default when this option is omitted.

-s, **--source** [!] *address*[/*mask*]
Source specification. *Address* can be either a hostname (please note that specifying any name to be resolved with a remote query such as DNS is a really bad idea), a network IPv6 address (with /mask), or a plain IPv6 address. (the network name isn't supported now). The *mask* can be either a network mask or a plain number, specifying the number of 1's at the left side of the network mask. Thus, a mask of *64* is equivalent to *ffff:ffff:ffff:ffff:0000:0000:0000:0000*. A "!" argument before the address specification inverts the sense of the address. The flag **--src** is an alias for this option.

-d, **--destination** [!] *address*[/*mask*]
Destination specification. See the description of the **-s** (source) flag for a detailed description of the syntax. The flag **--dst** is an alias for this option.

-j, **--jump** *target*
This specifies the target of the rule; i.e., what to do if the packet matches it. The target can be a user-defined chain (other than the one this rule is in), one of the special builtin targets which decide the fate of the packet immediately, or an extension (see **EXTENSIONS** below). If this option is omitted in a rule, then matching the rule will have no effect on the packet's fate, but the counters on the rule will be incremented.

-i, **--in-interface** [!] *name*
Name of an interface via which a packet is going to be received (only for packets entering the **INPUT**, **FORWARD** and **PREROUTING** chains). When the "!" argument is used before the interface name, the sense is inverted. If the interface name ends in a "+", then any interface which begins with this name will match. If this option is omitted, any interface name will match.

-o, **--out-interface** [!] *name*
Name of an interface via which a packet is going to be sent (for packets entering the **FORWARD** and **OUTPUT** chains). When the "!" argument is used before the interface name, the sense is inverted. If the interface name ends in a "+", then any interface which begins with this name will match. If this option is omitted, any interface name will match.

-c, **--set-counters PKTS BYTES** This enables the administrator to initialize the packet and byte counters of a rule (during **INSERT**, **APPEND**, **REPLACE** operations).

OTHER OPTIONS
The following additional options can be specified:

-v, **--verbose**
Verbose output. This option makes the list command show the interface name, the rule options (if any), and the TOS masks. The packet and byte counters are also listed, with the suffix 'K', 'M' or 'G' for 1000, 1,000,000 and 1,000,000,000 multipliers respectively (but see the **-x**

flag to change this). For appending, insertion, deletion and replacement, this causes detailed information on the rule or rules to be printed.

-n, --numeric
Numeric output. IP addresses and port numbers will be printed in numeric format. By default, the program will try to display them as host names, network names, or services (whenever applicable).

-x, --exact
Expand numbers. Display the exact value of the packet and byte counters, instead of only the rounded number in K's (multiples of 1000) M's (multiples of 1000K) or G's (multiples of 1000M). This option is only relevant for the **-L** command.

--line-numbers
When listing rules, add line numbers to the beginning of each rule, corresponding to that rule's position in the chain.

--modprobe=command
When adding or inserting rules into a chain, use **command** to load any necessary modules (targets, match extensions, etc).

MATCH EXTENSIONS
ip6tables can use extended packet matching modules. These are loaded in two ways: implicitly, when **-p** or **--protocol** is specified, or with the **-m** or **--match** options, followed by the matching module name; after these, various extra command line options become available, depending on the specific module. You can specify multiple extended match modules in one line, and you can use the **-h** or **--help** options after the module has been specified to receive help specific to that module.
The following are included in the base package, and most of these can be preceded by a **!** to invert the sense of the match.
tcp
These extensions are loaded if `--protocol tcp' is specified. It provides the following options:

--source-port [!] *port*[:*port*]
Source port or port range specification. This can either be a service name or a port number. An inclusive range can also be specified, using the format *port*:*port*. If the first port is omitted, "0" is assumed; if the last is omitted, "65535" is assumed. If the second port greater then the first they will be swapped. The flag **--sport** is a convenient alias for this option.

--destination-port [!] *port*[:*port*]
Destination port or port range specification. The flag **--dport** is a convenient alias for this option.

--tcp-flags [!] *mask comp*
Match when the TCP flags are as specified. The first argument is the flags which we should examine, written as a comma-separated list, and the second argument is a comma-separated list of flags which must be set. Flags are: **SYN ACK FIN RST URG PSH ALL NONE.** Hence the command

```
 ip6tables -A FORWARD -p tcp --tcp-flags SYN,ACK,FIN,RST SYN
```

will only match packets with the SYN flag set, and the ACK, FIN and RST flags unset.

[!] --syn
Only match TCP packets with the SYN bit set and the ACK and RST bits cleared. Such packets are used to request TCP connection initiation; for example, blocking such packets coming in an interface will prevent incoming TCP connections, but outgoing TCP connections will be unaffected. It is equivalent to **--tcp-flags SYN,RST,ACK SYN**. If the "!" flag precedes the "--syn", the sense of the option is inverted.

--tcp-option [!] *number*
Match if TCP option set.

udp
These extensions are loaded if `--protocol udp' is specified. It provides the following options:

--source-port [!] *port*[:*port*]
Source port or port range specification. See the description of the **--source-port** option of the TCP extension for details.

--destination-port [!] *port*[:*port*]
Destination port or port range specification. See the description of the **--destination-port** option of the TCP extension for details.

ipv6-icmp
This extension is loaded if `--protocol ipv6-icmp' or `--protocol icmpv6' is specified. It provides the following option:

--icmpv6-type [!] *typename*
This allows specification of the ICMP type, which can be a numeric IPv6-ICMP type, or one of the IPv6-ICMP type names shown by the command
 ip6tables -p ipv6-icmp -h

mac
--mac-source [!] *address*
Match source MAC address. It must be of the form XX:XX:XX:XX:XX:XX. Note that this only makes sense for packets coming from an Ethernet device and entering the **PREROUTING**, **FORWARD** or **INPUT** chains.

limit
This module matches at a limited rate using a token bucket filter. A rule using this extension will match until this limit is reached (unless the `!' flag is used). It can be used in combination with the **LOG** target to give limited logging, for example.

--limit *rate*
Maximum average matching rate: specified as a number, with an optional `/second', `/minute', `/hour', or `/day' suffix; the default is 3/hour.
--limit-burst *number*
Maximum initial number of packets to match: this number gets recharged by one every time the limit specified above is not reached, up to this number; the default is 5.

multiport
This module matches a set of source or destination ports. Up to 15 ports can be specified. It can only be used in conjunction with **-p tcp** or **-p udp**.
--source-ports *port*[,*port*[,*port*...]]

Match if the source port is one of the given ports. The flag **--sports** is a convenient alias for this option.
--destination-ports *port*[,*port*[,*port*...]]
Match if the destination port is one of the given ports. The flag **--dports** is a convenient alias for this option.
--ports *port*[,*port*[,*port*...]]
Match if the both the source and destination ports are equal to each other and to one of the given ports.

mark
This module matches the netfilter mark field associated with a packet (which can be set using the **MARK** target below).
--mark *value*[/*mask*]
Matches packets with the given unsigned mark value (if a mask is specified, this is logically ANDed with the mask before the comparison).

owner
This module attempts to match various characteristics of the packet creator, for locally-generated packets. It is only valid in the **OUTPUT** chain, and even this some packets (such as ICMP ping responses) may have no owner, and hence never match. This is regarded as experimental.
--uid-owner *userid*
Matches if the packet was created by a process with the given effective user id.
--gid-owner *groupid*
Matches if the packet was created by a process with the given effective group id.
--pid-owner *processid*
Matches if the packet was created by a process with the given process id.
--sid-owner *sessionid*
Matches if the packet was created by a process in the given session group.

TARGET EXTENSIONS
ip6tables can use extended target modules: the following are included in the standard distribution.
LOG
Turn on kernel logging of matching packets. When this option is set for a rule, the Linux kernel will print some information on all matching packets (like most IPv6 IPv6-header fields) via the kernel log (where it can be read with *dmesg* or *syslogd*(8)). This is a "non-terminating target", i.e. rule traversal continues at the next rule. So if you want to LOG the packets you refuse, use two separate rules with the same matching criteria, first using target LOG then DROP (or REJECT).
--log-level *level*
Level of logging (numeric or see *syslog.conf*(5)).
--log-prefix *prefix*
Prefix log messages with the specified prefix; up to 29 letters long, and useful for distinguishing messages in the logs.
--log-tcp-sequence
Log TCP sequence numbers. This is a security risk if the log is readable by users.
--log-tcp-options
Log options from the TCP packet header.
--log-ip-options

Log options from the IPv6 packet header.

MARK
This is used to set the netfilter mark value associated with the packet. It is only valid in the **mangle** table.
--set-mark *mark*

REJECT
This is used to send back an error packet in response to the matched packet: otherwise it is equivalent to **DROP** so it is a terminating TARGET, ending rule traversal. This target is only valid in the **INPUT**, **FORWARD** and **OUTPUT** chains, and user-defined chains which are only called from those chains. The following option controls the nature of the error packet returned:
--reject-with *type*
The type given can be
 icmp6-no-route
 no-route
 icmp6-adm-prohibited
 adm-prohibited
 icmp6-addr-unreachable
 addr-unreach
 icmp6-port-unreachable
 port-unreach
which return the appropriate IPv6-ICMP error message (**port-unreach** is the default). Finally, the option **tcp-reset** can be used on rules which only match the TCP protocol: this causes a TCP RST packet to be sent back. This is mainly useful for blocking *ident* (113/tcp) probes which frequently occur when sending mail to broken mail hosts (which won't accept your mail otherwise).

*** SNAT
This target is only valid in the **nat** table, in the POSTROUTING chain. It specifies that the source address of the packet should be modified (and all future packets in this connection will also be mangled), and rules should cease being examined. It takes one type of option:

```
     --to-source  ipaddr
```

which can specify a single new source IP address.

DIAGNOSTICS
Various error messages are printed to standard error. The exit code is 0 for correct functioning. Errors which appear to be caused by invalid or abused command line parameters cause an exit code of 2, and other errors cause an exit code of 1.
BUGS
Bugs? What's this? ;-) Well... the counters are not reliable on sparc64.

COMPATIBILITY WITH IPCHAINS
This **ip6tables** is very similar to ipchains by Rusty Russell. The main difference is that the chains **INPUT** and **OUTPUT** are only traversed for packets coming into the local host and originating from the local host

respectively. Hence every packet only passes through one of the three chains (except loopback traffic, which involves both INPUT and OUTPUT chains); previously a forwarded packet would pass through all three. The other main difference is that **-i** refers to the input interface; **-o** refers to the output interface, and both are available for packets entering the **FORWARD** chain. There are several other changes in ip6tables.

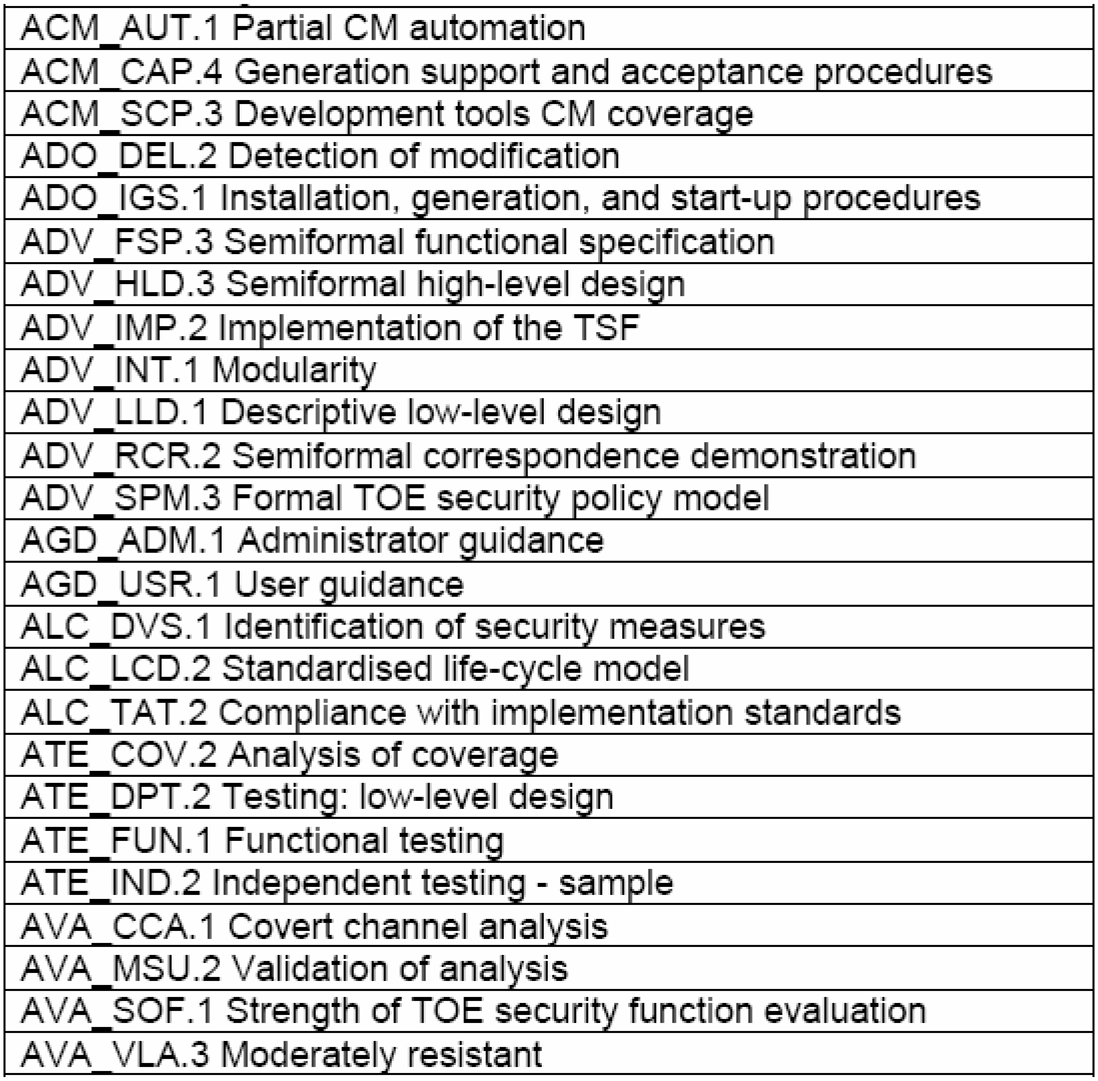
SEE ALSO

ip6tables-save(8), **ip6tables-restore(8)**, **iptables**(8), **iptables-save**(8), **iptables-restore**(8). The packet-filtering-HOWTO details iptables usage for packet filtering, the NAT-HOWTO details NAT, the netfilter-extensions-HOWTO details the extensions that are not in the standard distribution, and the netfilter-hacking-HOWTO details the netfilter internals.
See **http://www.netfilter.org/**.
AUTHORS
Rusty Russell wrote iptables, in early consultation with Michael Neuling.
Marc Boucher made Rusty abandon ipnatctl by lobbying for a generic packet selection framework in iptables, then wrote the mangle table, the owner match, the mark stuff, and ran around doing cool stuff everywhere.
James Morris wrote the TOS target, and tos match.
Jozsef Kadlecsik wrote the REJECT target.
Harald Welte wrote the ULOG target, TTL match+target and libipulog.
The Netfilter Core Team is: Marc Boucher, Martin Josefsson, Jozsef Kadlecsik, James Morris, Harald Welte and Rusty Russell.
ip6tables man page created by Andras Kis-Szabo, based on iptables man page written by Herve Eychenne
<rv@wallfire.org>.
*** ip6tables man page was modified by Trevor J. Baumgartner and Matthew D. W. Phillips to reflect added NAT functionality.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F. COMMON CRITERIA

This appendix contains a summary of the requirements necessary for an EAL5 certification. A listing of the requirements can be found in the following table.



ACM_AUT.1 Partial CM automation
ACM_CAP.4 Generation support and acceptance procedures
ACM_SCP.3 Development tools CM coverage
ADO_DEL.2 Detection of modification
ADO_IGS.1 Installation, generation, and start-up procedures
ADV_FSP.3 Semiformal functional specification
ADV_HLD.3 Semiformal high-level design
ADV_IMP.2 Implementation of the TSF
ADV_INT.1 Modularity
ADV_LLD.1 Descriptive low-level design
ADV_RCR.2 Semiformal correspondence demonstration
ADV_SPM.3 Formal TOE security policy model
AGD_ADM.1 Administrator guidance
AGD_USR.1 User guidance
ALC_DVS.1 Identification of security measures
ALC_LCD.2 Standardised life-cycle model
ALC_TAT.2 Compliance with implementation standards
ATE_COV.2 Analysis of coverage
ATE_DPT.2 Testing: low-level design
ATE_FUN.1 Functional testing
ATE_IND.2 Independent testing - sample
AVA_CCA.1 Covert channel analysis
AVA_MSU.2 Validation of analysis
AVA_SOF.1 Strength of TOE security function evaluation
AVA_VLA.3 Moderately resistant

Table 4. EAL5 Requirements

1. CONFIGURATION MANAGEMENT AUTOMATION

1.1 Partial CM Automation (ACM_AUT.1)

This component requires that the developer use and provide a CM plan. In addition, the CM system must provide an automated method through which only authorized changes are made to the TOE. The CM must also support the generation of the TOE. Finally, the CM plan must describe the automated tools used in the CM system and how the tools are used. [CC]

2. CONFIGURATION MANAGEMENT CAPABILITIES

2.1 Generation Support and Acceptance Procedures (ACM_CAP.4)

This component states that the developer must provide a reference for the TOE, use a CM system and provide CM documentation. In addition, the reference for the TOE must be unique to each version of the TOE and be labeled as such. The CM documentation must also include a configuration list, a CM plan and an acceptance plan. Within the configuration list, all configuration items that compromise the TOE must be uniquely identified and described. The CM system must also provide measures to ensure that only authorized changes are made to the configuration items, as well as support the generation of the TOE. [CC]

3. CONFIGURATION MANAGEMENT SCOPE

3.1 Development Tools CM Coverage (ACM_SCP.3)

This component requires the developer to provide a list of configuration management items for the TOE. This list must include implementation representation, security flaws, development tools and evaluation evidence required by the assurance components in the ST. [CC]

4. DELIVERY

4.1 Detection of Modification (ADO_DEL.2)

The developer must document and use procedures for delivery of the TOE or parts of it to the user. The documentation must describe all the procedures necessary to maintain security when distributing versions of the TOE to a user’s site. The documentation must also describe how the various procedures and technical measures provide for the detection of modifications, or any discrepancy between the developer’s master copy and the version received at the user’s site. [CC]

5. INSTALLATION, GENERATION AND START-UP

5.1 Installation, Generation, and Start-up Procedures (ADO_IGS.1)

This component requires the developer to document the procedures necessary for the secure installation, generation, and start-up of the TOE. [CC]

6. FUNCTIONAL SPECIFICATION

6.1 Semiformal Functional Specification (ADV_FSP.3)

This component states that the developer must provide a functional specification. The specification should describe the TSF using a semiformal style, supported by informal, explanatory text where appropriate. This specification must be internally consistent as well as completely represent the TSF. [CC]

7. HIGH-LEVEL DESIGN

7.1 Semiformal High-Level Design (ADV_HLD.3)

The high-level design requirements for developer action states that the developer must provide the high-level design of the TSF. This design should be semiformal and internally consistent. The design must also describe the structure of the TSF in terms of subsystems and the secure functionality provided within each subsystem. The high-level design should, in addition, identify any hardware, firmware or software required by the TSF and any underlying protection mechanisms. [CC]

8. IMPLEMENTATION REPRESENTATION

8.1 Implementation of the TSF (ADV_IMP.2)

The primary requirement for this component is for the developer to provide the implementation representation for the entire TSF. This representation must unambiguously define the TSF so that one would be able to recreate the implementation without making any design decisions. The representation should be internally consistent and describe the relationships between all portions of the implementation. [CC]

9. TSF INTERNALS

9.1 Modularity (ADV_INT.1)

This component states that the developer must design and structure the TSF in a modular fashion that avoids unnecessary interactions between the modules of the design. The developer must also provide an architectural description. The description must identify the modules of the TSF, describe the purpose of each module and describe how the TSF design provides for largely independent modules that avoid unnecessary interactions. [CC]

10. LOW-LEVEL DESIGN

10.1 Descriptive Low-Level Design (ADV_LLD.1)

This component requires the developer to provide an informal, low-level design of the TSF. This design must be internally consistent, describe the TSF in terms of modules, and describe the purpose of each module and its relationship between other modules. This design must identify all interfaces to the modules of the TSF and which modules are externally visible. The design must also describe the purpose and method of use for all modules within the TSF. [CC]

11. REPRESENTATION CORRESPONDENCE

11.1 Semiformal Correspondence Demonstration (ADV_RCR.2)

This requirement states that the developer must provide an analysis of correspondence between all adjacent pairs of TSF representations that are provided. These representations must demonstrate that all relevant security functionality is correctly and completely refined in the less abstract TSF representation. Also, a demonstration of correspondence between semiformal representations is required. [CC]

12. SECURITY POLICY MODELING

12.1 Formal TOE Security Policy Model (ADV_SPM.3)

The primary requirement for this component is for the developer to provide a formal TSP model. The developer must demonstrate correspondence between the functional specification and the TSP model. The TSP model must describe the rules and characteristics of all policies of the TSP that can be modeled. It must also include a demonstration of consistency and completeness with regards to all policies of the TSP. [CC]

13. ADMINISTRATOR GUIDANCE

13.1 Administrator Guidance (AGD_ADM.1)

This component requires the developer to provide administrator guidance addressed to system administrative personnel. The guidance should describe the administrative functions of the TOE and must be consistent with all other documentation supplied for evaluation. This guidance must also describe how to administer the TOE in a secure manner. [CC]

14. USER GUIDANCE

14.1 User Guidance (AGD_USR.1)

The primary requirement for this component is that user guidance be provided. This guidance must describe the functions and interfaces available to the non-administrative users of the TOE. It must also describe the use of user-accessible security functions provided by the TOE as well as any warnings that might occur. In addition the guidance must be consistent with all other documentation supplied for evaluation. [CC]

15. DEVELOPMENT SECURITY

15.1 Identification of Security Measures (ALC_DVS.1)

This component requires the developer to produce development security documentation. This documentation must describe all the physical, procedural, personnel and other security measures necessary to protect the confidentiality and integrity of the TOE design and implementation in its development environment. In addition, the documentation must also provide evidence that these security measures are followed during the development and maintenance of the TOE. [CC]

16. LIFE CYCLE DEFINITION

16.1 Standardized Life-Cycle Model (ALC_LCD.2)

The developer must establish and use a standardized life-cycle model to be used in the development and maintenance of the TOE. This life-cycle model implementation must also have corresponding documentation. This model must provide for the necessary control over the development and maintenance of the TOE. The life-cycle definition documentation must explain why the model was chosen and how it was used during development. [CC]

17. TOOLS AND TECHNIQUES

17.1 Compliance with Implementation Standards (ALC_TAT.2)

This component states that the developer must identify the development tools being used for the TOE. In addition, the implementation-dependent options of the development tools must be documented. Also, the development tools used in the implementation must be well defined. [CC]

18. COVERAGE

18.1 Analysis of Coverage (ATE_COV.2)

This component requires the developer to provide an analysis of the test coverage. The analysis must demonstrate the correspondence between the tests identified in the test documentation and the TSF as described in the functional specification. Also, the tests identified in the test documentation must be complete. [CC]

19. DEPTH

19.1 Testing: Low-Level Design (ATE_COV.2)

This component requires the developer to provide an analysis of the depth of testing. This analysis must demonstrate that the tests identified in the test documentation are sufficient to demonstrate that the TSF operates in accordance with its high and low level design. [CC]

20. FUNCTIONAL TESTS

20.1 Functional Testing (ATE_FUN.1)

This component requires the developer to test the TSF and document the results and provide test documentation. The documentation should consist of test plans, test procedure descriptions and actual test results. The testing procedure descriptions must identify the tests to be performed and describe the testing scenarios for testing each security function. The test results should demonstrate that each tested security function behaved as expected. [CC]

21. INDEPENDENT TESTING

21.1 Independent Testing – Sample (ATE_IND.2)

This component requires the developer to provide a suitable TOE for testing. The developer must also provide an equivalent set of resources to those that were used in the developer's functional testing of the TSF. [CC]

22. COVERT CHANNEL ANALYSIS

22.2 Covert Channel Analysis (AVA_CCA.1)

This component requires the developer to conduct a search for covert channels for each information flow control policy and provide analysis documentation. The documentation must identify covert channels and estimate their capacity. It must also describe the procedures used for determining the existence of covert channels. The documentation must also describe all assumptions made during the analysis as well as the method used for estimating channel capacity. It must also describe the worst case exploitation scenario for each identified covert channel. [CC]

23. MISUSE

23.3 Validation of Analysis (AVA_MSU.2)

This component requires the developer to provide guidance documentation as well as a document of the analysis of it. The guidance document must identify all possible modes of operation of the TOE, their consequences and implications for maintaining secure operation. The guidance document must list all assumptions about the intended environment as well as requirements for external security measures. [CC]

24. STRENGTH OF TOE SECURITY FUNCTIONS

24.1 Strength of TOE Security Function Evaluation (AVA_SOF.1)

This component requires the developer to perform a strength of TOE security function analysis for each mechanism identified in the ST as having a strength of TOE security function claim. Also, for each mechanism with a strength of TOE security function claim, the strength of the TOE security function analysis must show that it meets or exceeds the minimum strength level and the specific strength of function metric defined in the PP/ST. [CC]

25. VULNERABILITY ANALYSIS

25.1 Moderately Resistant (AVA_VLA.3)

This component requires the developer to perform a vulnerability analysis and provide documentation. This documentation must describe the analysis of the TOE deliverables performed to search for ways in which a user can violate the TSP. It must also describe the disposition of the identified vulnerabilities. Also, it must show that these vulnerabilities cannot be exploited in the specified environment for the TOE. In addition, the documentation must justify that the TOE is resistant to obvious penetration attacks. [CC]

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G. INSTALLATION GUIDE

This document is intended to guide the installation and setup of the modified 2.6.5 Linux kernel that supports NAT for IPv6. It also describes procedures for setting up the networking configurations for the TPE in order to run NAT. Due to the myriad of situations that may be encountered, this document only describes the basic steps needed and does not cover extenuating circumstances brought about by other machines.

1. Install Red Hat 9.0

2. Boot into the Red Hat 9.0 kernel

3. Verify network connectivity through an IPv4 ping

4. Insert NAT kernel CD and, if necessary, mount the CD

5. Copy the main tar file to /home:

```
# cp /mnt/cdrom/IPV6NAT.COMPLETE.tar /home
```

6. Remove the NAT kernel CD and, if necessary, unmount the CD

7. Change directory to the /home directory and then unpack the main tar archive:

```
# cd /home
# tar xfv IPV6NAT.COMPLETE.tar
```

8. This should produce three tar archives:
   - IPV6NAT.IPTABLES.tar ; contains the iptables user space code
   - IPV6NAT.MODUTILS.tar ; contains modutils necessary to compile 2.6 kernel.
   - IPV6NAT.KERNEL.tar ; contains the main kernel

9. Unpack the kernel archive:

```
# tar xfv IPV6NAT.KERNEL.tar
```

The main kernel directory is: /home/usagi/kernel/linux26/ assuming you unpacked the archives in the /home directory.

*If installation has been done previously, skip steps 10 through 19.

10. Unpack the modutils archive:

```
# tar xfv IPV6NAT.MODUTILS.tar
```

11. Change directory to the modutils folder:

```
# cd module-init-tools-0.9.15-pre4
```

12. Now modutils will be installed. For a more detailed installation guide, read the INSTALL file in the modutils main directory. The next step will be to configure the package for installation:

```
# ./configure --prefix=/
# make moveold
```

13. Next run make clean and make:

```
# make clean
# make
```

14. Then run make install:

```
# make install
# ./generate-modprobe.conf/etc/modprobe.conf
```

15. Change directory back to /home:

```
# cd ..  or cd /home
```

16. Unpack the iptables tar package:

```
# tar xfv IPV6NAT.IPTABLES.tar
```

17. Change directory into the iptables folder:

```
# cd iptables-1.2.9rc1
```

18. Now iptables will be installed. For a more detailed installation guide, read the INSTALL file in the modutils main directory. Run make, telling it where the kernel is located:

```
# make KERNEL_DIR=/home/usagi/kernel/linux26/
```

19. Run make install using the same information:

```
# make install KERNEL_DIR=/home/usagi/kernel/linux26
```

20. Change directory to the kernel directory:

```
# cd /home/usagi/kernel/linux26
```

21. At this point the kernel needs configuration. If installation done on a machine other than the TPE, the kernel will require reconfiguration. In the current directory is the kernel configuration used on the development machine. The configuration is named .config. To configure the kernel using this configuration run:

```
# make oldconfig
```

This should answer all of the kernel option questions.

22. To help generate the .config file on a different system:

```
# cd /usr/src/linux-2.4.20.8
# make mrproper
# make oldconfig
```

Answer all of the questions. It is highly recommended that the user have a solid knowledge of kernel configuration before starting. Improper configuration can lead to serious problems.

```
# cd /home/usagi/kernel/linux26/
# make mrproper
# make oldconfig
```

23. For platform specific questions, refer to the Linux 2.4.20.8 .config file. For configuration parameter that exist in both the Linux 2.4.20.8 and the Linux 2.6.5 .config files, use the Linux 2.6.5 .config file for reference. For configuration parameters that do not exist in Linux 2.4, use the Linux 2.6.5 .config file for reference. When in doubt, deny experimental modules and unknown drivers.

24. This step is only required if the kernel options are reconfigured and may be skipped if this is a TPE installation. The following is a list of kernel configuration options that must be enabled for NAT to function properly:

Networking Support (NET)
Packet Socket (PACKET)
TCP/IP Networking (INET)
IP6 tables support (IP6_NF_IPTABLES)

limit match support (IP6_NF_MATCH_LIMIT)

MAC address match support (IP6_NF_MATCH_MAC)

Routing header match support (IP6_NF_MATCH_RT)

Hop-by-hop and Dst opts header match support (IP6_NF_MATCH_OPTS)

Fragmentation header match support (IP6_NF_MATCH_FRAG)

HL match support (IP6_ NF_MATCH_HL)

Multiple port match support (IP6_NF_MATCH_MULTIPORT)

Owner match support (IP6_NF_MATCH_OWNER)

netfilter MARK match support (IP6_NF_MATCH_MARK)

IPv6 Extension Headers Match (IP6_NF_MATCH_IPV6HEADER)

AH/ESP match support (IP6_NF_MATCH_AHESP)

Packet Length match support (IP6_NF_MATCH_LENGTH)

EUI64 address check (IP6_NF_MATCH_EUI64)

Connection tracking (IP6_NF_CONNTRACK)

Connection state match support (IP6_NF_MATCH_STATE)

Packet filtering (IP6_NF_FILTER)

LOG target support (IP6_NF_TARGET_LOG)

REJECT target support (IP6_NF_TARGET_LOG)

Packet mangling (IP6_NF_MANGLE)

HL target support (IP6_NF_TARGET_HL)

MARK target support (IP6_NF_TARGET_MARK)

IP6 range match support (IP6_NF_MATCH_IPRANGE)

Full NAT (IP6_NF_NAT)

NETMAP target support (IP6_NF_TARGET_NETMAP)

SAME target support (IP6_NF_TARGET_SAME)

NAT of local connections (IP6_NF_NAT_LOCAL)

Network Packet Filtering (NETFILTER)

Connection Tracking (IP_NF_CONNTRACK)

IP Tables Support (IP_NF_IPTABLES)

Limit match support (IP_NF_MATCH_IPRANGE)

MAC address match support (IP_NF_MATCH_MAC)

Packet type match support (IP_NF_MATCH_PKTTYPE)

Netfilter mark match support (IP_NF_MATCH_MARK)

Multiple port match support (IP_NF_MATCH MULTIPORT)

TOS match support (IP_NF_MATCH TOS)

Recent match support (IP_NF_MATCH RECENT)

Length match support (IP_NF_MATCH LENGTH)

TTL match support (IP_NF_MATCH_TTL)

Connection state match support (IP_NF_MATCH_STATE)

Connection tracking match support

(IP_NFMATCH_CONNTRACK)
Owner match support (IP_ NF_MATCH_OWNER)
Packet filtering (IP_NF_FILTER)
Full NAT (IP_NF_NAT)
MASQUERADE target support
(IP_NF_TARGET_MASQUERADE)
REDIRECT target support (IP_NF_TARGET REDIRECT)
NETMAP target support (IP_NF_TARGET_NETMAP)
SAME target support (IP_NF_TARGET_SAME)
NAT of local connections (IP_NF_NAT_LOCAL)
Packet mangling (IP_NF_MANGLE)

25. Next step is to clear out already compiled object files:

```
# make clean
```

26. To add a specific tag to this compiled version of the kernel, bring up the Makefile located in the directory you are in and change the name from "IPv6-NAT" to whatever tag you like. "IPv6-NAT" is the default tag.

27. Compile the kernel:

```
# make bzImage
```

28. Make the modules:

```
# make modules
```

29. Install the modules:

```
# make modules_install
```

30. Run install:

```
# make install
```

31. Bring up the grub.conf file and edit it. Change directory to /boot/grub and then bring up grub.conf in the editor of your choice.

32. Edit the line just below your kernel label that says root=LABEL=/. Change root=LABEL=/ to say root=/dev/hda2.
*NOTE: this is a configuration issue that may not be present on other machines and the hda2 label can change from machine to machine. This change specifically sets up the kernel for the TPE.

33. Reboot and select the NAT kernel from the grub list.

34. Check network connectivity through an IPv4 ping

/* NAT SETUP */

35. Once logged in, bring up a terminal window

36. Issuing the following commands will setup both network interfaces. The global addresses may be changed, but the subnet of the internal computers must be the same.

```
# ifconfig eth0 inet6 add 2003::1/64
# ifconfig eth1 inet6 add 2004::1/64
```

37. Setup the user-space ip6tables:

```
# ip6tables -t nat -A POSTROUTING -o eth1 -j SNAT --to-source 2004::1
```

This assumes the same topography as the IPv6 testbed.

38. Turn on forwarding:

```
# sysctl -w net.ipv6.conf.all.forwarding=1
```

39. Verify IPv6 network connectivity through an IPv6 ping

40. NAT is now ready and functioning. All messages sent from the Client will be translated before being forwarded to the server, so that the server only sees the translated address.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. George Bieber
   OSD
   Washington, DC

4. RADM Joseph Burns
   Fort George Meade, MD

5. Deborah Cooper
   DC Associates, LLC
   Roslyn, VA

6. CDR Daniel L. Currie
   PMW 161
   San Diego, CA

7. LCDR James Downey
   NAVSEA
   Washington, DC

8. Richard Hale
   DISA
   Falls Church, VA

9. LCDR Scott Heller
   SPAWAR
   San Diego, CA

10. Wiley Jones
    OSD
    Washington, DC

11. Russell Jones
    N641
    Arlington, VA

12. David Ladd
Microsoft Corporation
Redmond, WA

13. Dr. Carl Landwehr
National Science Foundation
Arlington, VA

14. Steve LaFountain
NSA
Fort Meade, MD

15. Dr. Greg Larson
IDA
Alexandria, VA

16. Ray A. Letteer
Head, Information Assurance, HQMC C4 Directorate
Washington, DC

17. Penny Lehtola
NSA
Fort Meade, MD

18. Ernest Lucier
Federal Aviation Administration
Washington, DC

19. CAPT Sheila McCoy
Headquarters U.S. Navy
Arlington, VA

20. Dr. Ernest McDuffie
National Science Foundation
Arlington, VA

21. Dr. Vic Maconachy
NSA
Fort Meade, MD

22. Doug Maughan
Department of Homeland Security
Washington, DC

23. Dr. John Monastra
Aerospace Corporation
Chantilly, VA

24. John Milder
SPAWAR
Charleston, SC

25. Marshall Potter
Federal Aviation Administration
Washington, DC

26. Dr. Roger R. Schell
Aesec
Pacific Grove, CA

27. Keith Schwalm
Good Harbor Consulting, LLC
Washington, DC

28. Dr. Ralph Wachter
ONR
Arlington, VA

29. David Wirth
N641
Arlington, VA

30. Daniel Wolf
NSA
Fort Meade, MD

31. CAPT Robert Zellman
CNO Staff N614
Arlington, VA

32. Dr. Kay G. Schulze
United States Naval Academy
Annapolis, MD

33. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, CA

34. Thuy D. Nguyen
Naval Postgraduate School
Monterey, CA

35. Trevor J. Baumgartner
Student, Naval Postgraduate School
Monterey, CA

36. Matthew D.W. Phillips
Student, Naval Postgraduate School
Monterey, CA